MASTER EN SCIENCES INFORMATIQUES

Étude des méthodes d'estimation du coût de développement de logiciels, en particulier la méthode des points de fonction

Albrecq, Jean-Marc; Belle, Jean-Marie

Award date:
1988

Awarding institution:
Universite de Namur

Etude des méthodes d'estimation du coût de développement de logiciels, en particulier la méthode des points de fonction.

Jean-Marc ALBRECQ
Jean-Marie BELLE.

Mémoire présenté en vue de l'obtention du grade de Licencié et Maître en Informatique.

Promoteur : Mr. Baudouin LE CHARLIER.

Année académique 1987-1988.

Remerciements

L'élaboration de ce mémoire a nécessité le concours de nombreuses personnes qui ont contribué à former une atmosphère propice à la recherche et à l'étude.

Nous tenons tout particulièrement à exprimer notre profonde gratitude à l'égard de Baudouin Le Charlier, promoteur de ce travail, pour les précieux conseils qu'il nous a prodigués lors de fréquents entretiens.

Nous adressons également nos plus vifs remerciements à Marie-Claire Verlaine, chef du service Méthodologie et Administration de l'Information à la Caisse Générale d'Epargne et de Retraite, pour nous avoir si aimablement intégrés à son équipe.

Nous éprouvons tout particulièrement de la reconnaissance envers Christian Graas, Alain Spiltoir, Eric van Renterghem et Gérard Magérat qui ont collaboré à ce travail. Ils nous ont consacré, avec beaucoup de dévouement, de nombreuses heures de leur temps précieux.

Enfin, nous remercions tous les membres du service MAI ainsi que toutes les personnes de la C.G.E.R. qui, de près ou de loin, ont contribué à la réalisation de notre projet. Toutes ces personnes nous ont par ailleurs laissé un excellent souvenir de notre première prise de contact avec un environnement professionnel.

ABSTRACT.

CE TRAVAIL ABORDE L'ASPECT ECONOMIQUE DU DEVELOPPEMENT DE LOGICIELS. POUR CE FAIRE, UNE ETUDE DETAILLEE DES FACTEURS QUI INFLUENCENT LA PRODUCTIVITE A ETE REALISEE. NOUS AVONS EXAMINE QUEL ETAIT L'ETAT DE L'ART DANS LES METHODES D'ESTIMATION : ETUDE DES METHODES D'HALSTEAD, WALSTON ET FELIX, PUTNAM ET BOEHM. NOUS AVONS ABORDE PLUS PARTICULIEREMENT LA METHODE DES POINTS DE FONCTION PROPOSEE PAR ALBRECHT AFIN DE L'ADAPTER POUR REALISER DES ESTIMATIONS A PRIORI PAR L'INTRODUCTION D'UN DIAGRAMME DE FLUX COMME ENTREE DE NOTRE IMPLEMENTATION.

THIS PAPER APPROACHES THE ECONOMICAL ASPECT OF SOFTWARE ENGINEERING. TO DO THAT, A DETAILED STUDY OF "COST DRIVERS" WHICH HAVE AN INFLUENCE ON THE PRODUCTIVITY HAS BEEN REALIZED. WE HAVE EXAMINATED WHAT IS THE STATE OF THE ART IN THE COST ESTIMATION METHODS : THE METHODS OF HALSTEAD, WALSTON AND FELIX, PUTNAM AND BOEHM. IN PARTICULAR, WE HAVE APPROACHED THE FUNCTION POINTS METHOD OF ALBRECHT TO ADAPT IT AND TO REALIZE AN ESTIMATION IN THE ALL FIRST STEPS OF THE LIFE CYCLE. WE DO THAT BY THE INTRODUCTION OF A FLOWCHART.

INTRODUCTION.

PARTIE 1
LA PROBLEMATIQUE LIEE AUX METHODES D'ESTIMATION.

CHAPITRE 1.
COMMENT EXPLIQUER L'INTERET NOUVEAU POUR LES METHODES D'ESTIMATION ?

1. Le contexte économique. 2.
2. Les intérêts. 3.

CHAPITRE 2.
QUE FAUT-IL MESURER OU ESTIMER ?

1. Quelques concepts à définir. 6.
   1.1. La productivité. 6.
   1.2. La prévision de la durée et du coût de développement. 6.
2. Les facteurs de coût. 7.
   2.1. Introduction. 7.
   2.2. Les facteurs liés au logiciel. 9.
      2.2.1. La taille du logiciel. 9.
         2.2.1.1. Le nombre de LOC 10.
         2.2.1.2. Le nombre d'opérateurs et opérandes d'Halstead 11.
         2.2.1.3. Le nombre de fonctionnalités requises par l'utilisateur 11.
      2.2.2. La qualité des logiciels. 15.
      2.2.3. La taille de la base de données. 17.
      2.2.4. La complexité du logiciel. 17.
      2.2.5. La quantité de documentation. 18.
      2.2.6. Le type d'application. 19.
      2.2.7. La réutilisabilité. 21.
   2.3. Les facteurs liés aux moyens de développement. 22.
      2.3.1. Le temps d'exécution, le temps de réponse, la capacité mémoire. 22.
      2.3.2. L'utilisation de moyens d'aide. 23.
      2.3.3. L'utilisation de techniques modernes de programmation. 24.
   2.4. Les facteurs liés au personnel. 26.
      2.4.1. La capacité du personnel. 27.
      2.4.2. L'expérience du personnel. 28.
      2.4.3. La baisse du nombre de spécialistes. 29.
   2.5. Les facteurs liés au logiciel. 30.
      2.5.1. Les exigences posées sur la durée de développement. 30.

2.5.2. La gestion du projet. 31.
2.6. Les facteurs liés aux utilisateurs. 32.
3. Conclusion. 33.
3.1. Le manque de consensus dans la terminologie utilisée. 33.
3.2. Une médiocre définition de la qualité d'un logiciel. 33.
3.3. La pauvre qualité des données concernant les coûts. 34.
3.4. La nature dynamique du domaine. 34.
3.5. L'aspect non quantitatif de certains facteurs. 34.
3.6. L'objectivité. 35.
3.7. La corrélation. 35.
3.8. L'estimation. 35.

CHAPITRE 3.
QUAND EST-IL OPPORTUN DANS LE CYCLE DE VIE D'UN PROJET DE RECOURIR A DES ESTIMATIONS ?

1. Introduction. 37.
2. Description simplifiée du modèle de cycle de vie défini à la C.G.E.R. 37.
2.1. La définition de projet. 38.
2.1.1. L'étude de l'existant. 38.
2.1.2. Les objectifs et contraintes du nouveau système. 38.
2.1.3. Les solutions possibles. 38.
2.2. L'analyse conceptuelle. 39.
2.2.1. L'analyse conceptuelle des données. 39.
2.2.2. L'analyse conceptuelle des traitements. 39.
2.2.2.1. Le modèle de décomposition des traitements. 39.
2.2.2.2. Le modèle de description d'un élément de la décomposition. 40.
2.2.2.3. Le modèle dynamique. 41.
2.3. L'analyse fonctionnelle. 41.
2.3.1. L'analyse fonctionnelle des traitements. 41.
2.3.2. L'analyse fonctionnelle des données. 42.
2.4. L'analyse technique. 42.
2.5. La programmation. 42.
2.6. Les tests. 42.
2.7. Le passage en production. 43.
2.8. La maintenance. 43.
3. Le moment idéal pour estimer la durée et le coût de développement d'un logiciel. 43.

CHAPITRE 4.
QUEL EST L'ETAT DE L'ART DANS LE DOMAINE DES METHODES D'ESTIMATION ?

1. Introduction. 46.
2. Les méthodes de conception de modèles d'estimation. 46.
2.1. L'expérimentation. 46.
2.2. L'analyse des données de projets terminés. 47.
3. Présentation d'une méthode d'estimation pour de petits projets (la méthode d'Halstead). 48.
4. Les méthodes d'estimation pour les grands projets. 49.

4.1.Les risques des méthodes intuitives réalisées par analogie et selon l'expérience. 49.
4.1.1. Les bases de données. 50.
4.1.2. La méthode par analogie. 51.
4.1.3. Le jugement expert. 51.
4.1.3.1. La méthode Delphi. 51.
4.1.3.2. Plusieurs estimations (la méthode de Putnam). 52.
4.2. Les premiers modèles algorithmiques. 52.
4.2.1. Le modèle de Nelson (modèle linéaire). 53.
4.2.2. Le modèle de Wolverton (modèle bottom-up). 55.
4.3. Les modèles les plus récents. 56.
4.3.1. La formule générale. 56.
4.3.2. Le modèle de Walston et Félix. 58.
4.3.3. Le modèle Cocomo de Boehm. 61.
4.3.4. Le modèle de Putnam/Norden. 62.
4.3.5. Le modèle de DeMarco 65.
4.3.6. La méthode des points de fonction. 66.
5. La distribution de la force de travail dans le temps. 67.
6. Conclusion. 68.

CHAPITRE 5.
PRESENTATION DES METHODES D'ESTIMATION LES PLUS COURANTES.

1. Introduction. 71.
2. La méthode de Walston et Félix. 71.
2.1. Motivations. 71.
2.2. Explication du modèle. 72.
3. La méthode d'Halstead. 83.
3.1. Motivations. 83.
3.2. Explication du modèle. 83.
3.2.1. La relation entre longueur et vocabulaire 86.
3.2.2. La taille du programme. 86.
3.2.3. Volume potentiel (V*) 87.
3.2.4. L'effort (E). 88.
3.2.5. Le temps de developpement (T). 89.
4. La méthode de L.H. Putnam 89.
4.1. Motivations. 89.
4.2. Explication du modèle. 90.
4.2.1. L'estimation de la taille. 92.
4.2.1.1. Dans la définition de projet. 92.
4.2.1.2. Dans l'analyse conceptuelle. 93.
4.2.1.3. Dans l'analyse fonctionnelle. 93.
4.2.2. La conversion de l'estimation de la taille en une estimation du temps, de l'effort et du coût. 94.
4.2.2.1. La détermination du temps et de l'effort. 94.
4.2.2.2 La détermination du coût. 95.
4.2.3. La détermination de la distribution optimale des personnes à tout moment du développement. 96.
5. La méthode de Boehm (le modèle cocomo). 98.
5.1. Motivations. 98.
5.2. Explication du modèle. 98.
5.2.1. Le modèle de base (macro-estimation). 99.

5.2.2. Le modèle intermédiaire. 99.
5.2.2.1. Première étape : l'estimation de l'effort nominal selon le mode de développement. 100.
5.2.2.2. Deuxième étape : les multiplicateurs d'effort. 100.
5.2.2.3. Troisième étape : l'estimation de l'effort de développement. 102.
5.2.2.4. Quatrième étape : les calculs additionnels. 103.
5.2.3. Le modèle détaillé. 103.
5.2.3.1. Première étape : la taille totale du système et l'effort nominal. 103.
5.2.3.2. Deuxième étape : l'influence des facteurs aux différents niveaux de la décomposition. 103.
6. Conclusion. 105.

CHAPITRE 6.
CONCLUSION GENERALE DE LA PREMIERE PARTIE.

PARTIE 2
LA METHODE DES POINTS DE FONCTION.

CHAPITRE 1.
LES BASES DE LA METHODE.

1. Les motivations. 110.
2. Le modèle d'application utilisé dans la méthode. 111.
2.1. Introduction. 111.
2.2. Définition d'une fonction d'entrée externe (input). 113.
2.3. Définition d'une fonction de sortie externe (output). 113.
2.4. Définition d'une fonction d'interface externe (interface). 113.
2.5. Définition d'une fonction de groupe de données logique interne (master file). 114.
2.6. Définition de la fonction de requête externe (inquiry). 114.
2.7. La notion de complexité d'une fonction. 114.
2.8. Attribution du niveau de complexité. 115.
3. Les principes de fonctionnement de la méthode. 115.
3.1. Le nombre de points de fonction brut. 115.
3.2. Le nombre de points de fonction net. 118.

CHAPITRE 2.
DIRECTIVES POUR DENOMBRER LES TYPES DE FONCTIONS ET LEUR ATTRIBUER UN NIVEAU DE COMPLEXITE.

1. Les inputs. 120.
1.1. Les niveaux de complexité. 120.
1.2. Corrections en raison de considérations techniques. 121.
1.3. Quelques inputs possibles. 122.
1.4. Conseils pour le comptage. 122.
2. Les outputs. 123.
2.1. Les niveaux de complexité. 123.
2.2. Corrections en raison de considérations techniques. 124.
2.3. Quelques outputs possibles. 125.
2.4. Conseils pour le comptage 125.
3. Les master files. 125.
3.1. Les niveaux de complexité. 126.
3.2. Corrections en raison de considérations techniques. 126.
3.3. Les master files possibles. 127.
3.4. Conseils pour le comptage. 127.
4. Les interfaces. 127.
4.1. Les niveaux de complexité. 127.
4.2. Corrections en raison de considérations techniques. 128.
4.3. Les interfaces possibles. 128.
4.4. Conseils pour le comptage. 128.
5. Les inquiries 129.
5.1. Les niveaux de complexité. 129.
5.2. Les inquiries possibles. 129.
5.3. Conseils pour le comptage. 130.
6. Les 14 caractéristiques ou facteurs d'ajustement. 130.
6.1. Télécommunication. 130.
6.2. Données distribuées ou traitements distribués. 130.
6.3. Performances. 131.
6.4. Charge. 131.
6.5. Volume de transactions. 131.
6.6. Data-entry. 131.
6.7. Aspect conversationnel de l'application. 131.
6.8. Mise-à-jour de données en "on-line". 131.
6.9. Traitements internes complexes. 132.
6.10. Réutilisabilité. 132.
6.11. La mise en service. 132.
6.12. Commodités de service. 132.
6.13. Localisation-organisation. 132.
6.14. Flexibilité. 132.

CHAPITRE 3.
CONSEILS PRATIQUES.

1. Introduction. 135.
2. Directives générales. 135.
3. Inputs. 135.
4. Outputs. 136.
5. Les groupes de données logiques internes (MF) 137.
6. Les interfaces. 138.
7. Les inquiries. 138.
8. Les facteurs de correction. 139.
8.1. Télécommunication. 139.
8.2. Données distribuées ou traitements distribués. 140.
8.3. Performances. 140.
8.4. Charge. 140.

8.5. Volume de transactions. 141.
8.6. Data-entry. 141.
8.7. Aspect conversationnel de l'application. 141.
8.8. Mise-à-jour des données en "on-line". 142.
8.9. Traitements internes complexes. 142.
8.10. Réutilisabilité. 142.
8.11. La mise en service. 143.
8.12. Commodités de service. 143.
8.13. Localisation-organisation. 144.
8.14. Flexibilité. 144.

CHAPITRE 4.
JUGEMENT DE LA METHODE DES POINTS DE FONCTION.

1. La définition. 146.
2. La fiabilité. 146.
3. L'objectivité. 146.
4. La compréhensibilité. 147.
5. Les détails. 147.
6. La stabilité. 148.
7. Le domaine d'application. 148.
8. La facilité d'utilisation. 148.
9. L'aspect prévisionnel. 148.
10. La pertinence et la complétude du choix des facteurs. 149.
11. La rapidité versus la qualité d'estimation. 149.
12. La possibilité d'améliorer l'estimation avec l'expérience. 149.
13. L'indépendance vis à vis de son environnement d'expérimentation. 150.

PARTIE 3
ADAPTATION ET IMPLEMENTATION DE LA METHODE DES POINTS DE FONCTION.

CHAPITRE 1.
INTRODUCTION

1. L'objectif du stage. 152.
2. Démarche suivie. 152.
3. Description générale de la vente de TC. 153.

CHAPITRE 2.
L'IMPLEMENTATION DE LA METHODE DES PF.

1. Motivations. 156.
2. Notre modèle d'application. 156.
2.1. Introduction. 156.
2.2. Les modifications. 157.
2.3. Les principes de fonctionnement du programme. 162.

2.4. Le diagramme de flux. 164.
2.4.1. Règles de représentation. 165.
2.4.2. Le formalisme. 166.
2.4.3. Les avantages du diagramme de flux. 167.
2.5. La grammaire du programme. 167.
2.6. Remarques. 169.
2.7. Mode d'emploi du programme. 169.
3. Estimation de l'effort et de la productivité à partir d'un nombre de PF. 170.
3.1. Une première estimation. 170.
3.2. Raffinement des estimations. 172.

CONCLUSION.

Introduction

Après quarante années d'automatisation, il est étonnant de constater que les instruments économiques ont fait à peine leur apparition dans le processus de développement de logiciel. Les décisions concernant les investissements et le temps consacrés au développement de projet sont le plus souvent basées sur l'intuition. Chacun se fie à sa propre expérience pour arriver à estimer la taille, la durée et le coût de développement d'un projet. Cette méthode semble bien fonctionner pour de petits projets. Cependant, si le système est grand et compliqué, l'intuition semble ne plus suffir.

Les dangers d'une planification trop optimiste pour le développement d'un nouveau logiciel ne sont pas seulement une méfiance accrue de la part des employeurs et/ou de l'utilisateur final, mais aussi un dépassement de budget, une relation faussée entre les coûts et les bénéfices et des coûts supplémentaires durant la phase de maintenance.

Par conséquent, il semble nécessaire d'élaborer des modèles quantitatifs susceptibles d'estimer ces paramètres le plus précisément et le plus rapidement possible. C'est dans ce contexte et dans le cadre de notre stage effectué à la Caisse Générale d'Epargne et de Retraite (C.G.E.R.) que l'on nous a demandé d'étudier un modèle d'estimation particulier : la méthode des points de fonction proposée par Allan Albrecht en 1979. La C.G.E.R. désire utiliser ce modèle pour réaliser des estimations en début de développement.

Notre point de départ fut l'article de R. van Straten [van Straten 87]. Cependant, à l'aide de ce seul article, il nous était impossible de poser un jugement sur la validité et l'originalité de cette méthode par rapport aux autres, car nous n'avions jamais abordé l'aspect économique du développement de logiciel durant nos études; par conséquent, nous étions tout à fait profanes en la matière. Nous avons élargi le cadre de nos investigations afin d'aborder l'aspect économique du génie logiciel. C'est ainsi qu'après de nombreuses recherches bibliographiques, nous avons étudié les méthodes d'estimation les plus connues pour mieux situer la méthode des points de fonction parmi celles-ci.

C'est aussi pour cette raison que nous subdiviserons notre travail en trois parties. Nous séparerons la partie traitant de la problématique des méthodes d'estimation (partie 1) de celles relatives à l'étude proprement dite des points de fonction (partie 2) et à son adaptation (partie 3).

En première partie, pour cerner la problématique des méthodes d'estimation, nous poserons quatre questions :

- **comment expliquer l'intérêt nouveau pour les méthodes d'estimation ? (le pourquoi).** En effet, pour notre bonne compréhension du cadre de travail, nous pensons qu'il est nécessaire de considérer les motivations qui sont à l'origine des méthodes d'estimation.

- **que faut-il estimer ou mesurer ? (le quoi).** Il est nécessaire de savoir précisément ce qu'estiment ces

méthodes. Nous en profiterons pour définir les concepts utilisés tout au long du travail. Par la suite, nous consacrerons une partie importante de notre étude aux facteurs de coût, c'est-à-dire aux caractéristiques des projets qui ont une influence sur l'effort à consacrer au développement. Ces facteurs sont en fait la base de toute méthode d'estimation et jouent par conséquent un rôle fondamental.

- **quand est-il opportun dans le cycle de vie d'un projet de recourir à des estimations ? (le quand).** Nous définirons ce que nous entendons par cycle de vie, étant donné la diversité des définitions proposées dans la littérature. Ceci est d'autant plus nécessaire que notre objectif est non seulement l'étude de la méthode d'Albrecht, mais aussi l'étude de son applicabilité en début de projet.

- **quel est l'état de l'art dans le domaine des méthodes d'estimation ? (le comment).** Nous y tracerons l'évolution historique des méthodes, tout en mettant en évidence les problèmes qui y sont liés.

Vu l'aspect fort général de cette dernière question, nous détaillerons le fonctionnement des méthodes les plus courantes.

La deuxième partie de ce travail sera l'étude approfondie de la méthode des points de fonction dans laquelle nous dégageons les bases de la méthode, les principes de fonctionnement et quelques conseils pratiques pour l'utiliser. Nous procéderons enfin à un jugement de la méthode.

Dans la dernière partie, nous décrirons notre adaptation et notre implémentation de la méthode des points de fonction afin de l'utiliser dans les premières phases du développement.

Démarche suivie.

Ce travail est le fruit d'une recherche bibliographique importante. Le plus souvent, nous avons procédé à la compilation de plusieurs articles. En particulier pour les facteurs de coûts, nous sommes partis de la classification proposée par Boehm [Boehm 81] que nous avons d'une part complétée, et d'autre part détaillée par des informations tirées d'autres auteurs. Pour tracer l'évolution des méthodes d'estimation, nous avons repris l'historique de van Vliet [van Vliet 87] que nous avons également précisée en consultant les papiers originaux des auteurs des méthodes.

Pour l'étude de la méthode des points de fonction, la documentation était difficilement disponible et de plus restait

très vague. La synthèse que nous avons réalisée à partir d'Albrecht [Albrecht 79], Rudolph [Rudolph 83], van Straten [van Straten 87] et Audit Informatica [Audit **], comprend néanmoins un apport personnel important afin d'éclaircir les concepts utilisés dans la méthode des points de fonction.

La troisième partie est une adaptation personnelle de la méthode.

Partie 1

La problématique liée aux méthodes d'estimation.

CHAPITRE 1.

COMMENT EXPLIQUER L'INTERET NOUVEAU POUR LES METHODES D'ESTIMATION ?

Dans la question du "pourquoi mesurer", nous avons distingué deux parties; la première sera consacrée à la description du contexte économique qui a suscité le désir de mesurer les coûts de développement de projets informatiques, la deuxième traitera des différents intérêts que l'on peut retirer de l'estimation des coûts d'un projet.

1. Le contexte économique.

Ces dernières années, le développement de projets informatiques a pris une expansion sans précédent; il est devenu un véritable créneau industriel pour faire face au nombre de programmes de plus en plus important. Par exemple, aux Etats Unis en 1980, le coût global de développement et d'acquisition de programmes s'élevait à 40 milliards de dollars, c'est-à-dire plus ou moins 2% du produit national brut [Boehm 81].

Les coûts résultant du développement et de la maintenance de projets n'arrêtent pas de croître depuis une trentaine d'années, alors que les coûts hardware suivent le chemin inverse. Ceci est illustré dans la figure 1 par Boehm.

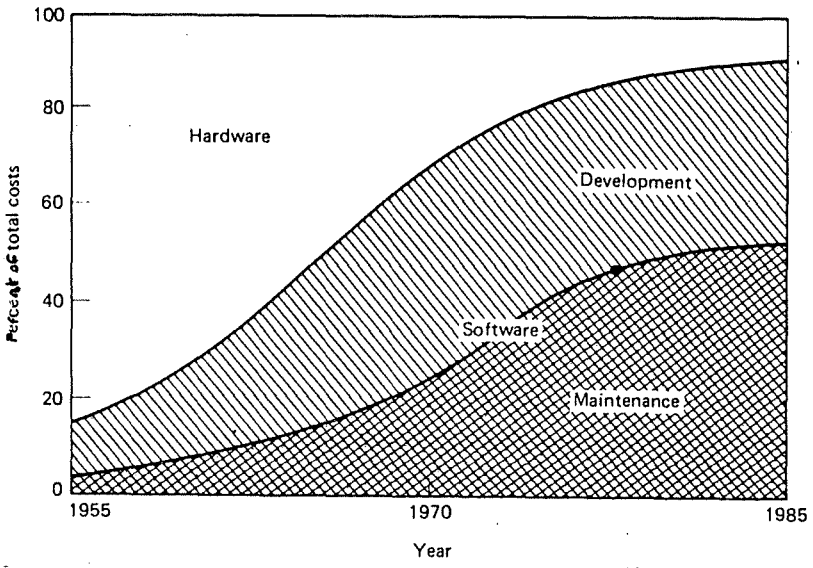


FIGURE 1 TENDANCES DES COUTS HARDWARE/SOFTWARE

En 1955, les coûts hardware représentaient environ 85% du coût global d'un projet alors que ceux du software (développement et maintenance) ne s'élevaient qu'à 15 %. Trente

années plus tard, on assista à un réel renversement de situation avec des coûts hardware ne représentant plus que 10 % du coût global et des coûts software représentant 90 %.

Nous pouvons expliquer ce phénomène par l'effet combiné d'une chute des prix sur le marché du hardware et du degré de complexité et de qualité requis pour les applications actuelles.

De plus, le développement de logiciel est une activité orientée homme nécessitant une main-d'oeuvre spécialisée et coûteuse pour assurer ce degré de complexité et de qualité. Cette activité de développement suit les mêmes objectifs que toute activité économique, à savoir maximiser le profit en minimisant les coûts.

2. Les intérêts.

Les entreprises accordent un intérêt primordial aux mesures de productivité de l'activité de développement car elles pourraient ainsi mieux cerner le comportement de cette activité pour aboutir à une meilleure compréhension et à une meilleure gestion de celle-ci. Nolan disait déjà en 1979 : "Ce que vous ne pouvez mesurer, vous ne pouvez le maîtriser" [Green 85]. de Kater ajouta : "Sans mesure précise de productivité, il est très difficile de déterminer la valeur d'un système de production dans le temps et de fixer l'utilité relative des outils et des méthodologies de développement" [de Kater 87].

Par une meilleure compréhension de l'activité de développement obtenue en analysant les écarts de productivité dans les anciens projets, on pourrait dénombrer les ressources humaines, matérielles et le temps requis pour les différentes étapes du développement. Cette connaissance permettrait d'arriver à une meilleure planification favorisant le respect des échéances et des budgets fixés. Selon une enquête effectuée par Greene pour le compte du gouvernement britannique en 1984, sur deux cents applications de tous types, 66 % d'entre elles ont dépassé l'échéance fixée, 55 % ont dépassé les coûts, 45 % avaient un degré de complexité plus important que prévu et 53 % d'entre elles confrontaient les développeurs à des problèmes sérieux et inattendus.

Ces chiffres montrent bien l'utilité des méthodes d'estimation qui renferment dans leur modèle les fruits de la meilleure compréhension du processus de développement.

De plus, ces méthodes d'estimation offrent des avantages supplémentaires :

- elles permettent de comparer les coûts de différentes solutions. Par exemple, estimer combien coûterait l'introduction de nouvelles caractéristiques dans le projet telles qu'un niveau

de fiabilité supérieure ou une fonctionnalité supplémentaire demandée par l'utilisateur.

- elles permettent également de réaliser un suivi du niveau de productivité dans le temps (au sein d'une même organisation).

Par conséquent, une méthode d'estimation est un véritable outil d'aide à la gestion d'un projet informatique.

CHAPITRE 2.

QUE FAUT-IL MESURER

OU ESTIMER ?

1. Quelques concepts à définir.

Si l'on s'en tient à l'évolution historique, les recherches se sont d'abord orientées vers le calcul de la productivité pour les raisons économiques évoquées précédemment. Par la suite, étant donné le nombre croissant de projets, il est paru nécessaire d'estimer le plus tôt possible la durée et le coût du développement.

1.1. La productivité.

Nous définirons la productivité comme étant le rapport entre le résultat obtenu et l'effort requis pour l'obtenir.

En matière de développement de projets informatiques, on définit le résultat obtenu par la taille du projet exprimée en lignes de code (noté LOC) et l'effort par le nombre de mois-homme (noté M-H) requis pour le développement. Par conséquent, l'unité de productivité la plus souvent rencontrée dans les ouvrages est le nombre de LOC produites par mois-homme, ce que l'on note LOC/M-H.

Selon Boehm dans [Boehm 81], un M-H équivaut à 152 heures de travail. Quant à la définition d'une LOC, elle ne fait pas l'unanimité car certains auteurs incluent dans ce concept le texte donné en commentaire, les lignes blanches, alors que d'autres ne considèrent que les instructions écrites dans le langage de programmation employé.

1.2. La prévision de la durée et du coût de développement.

Par durée de développement, on entend le temps qui s'écoule entre le début du projet et sa livraison.

Par coût de développement, on entend l'effort requis à la production du projet exprimé en M-H et traduit en francs. Cette traduction en francs n'étant qu'une simple multiplication (le nombre de M-H multiplié par la valeur d'un M-H dans l'organisation considérée), on se contentera dans la suite du travail de parler de coût quantitatif (effort en M-H) plutôt que de coût en francs.

Intuitivement, on pourrait penser que l'effort requis pour le développement ne dépend que de la difficulté intrinsèque du problème. Cependant, nous verrons dans la section suivante qu'il existe une liste importante de facteurs de coût qui influent sur l'effort.

2. Les facteurs de coût.

2.1. Introduction.

De nombreux facteurs influencent le coût d'un logiciel, Noth en a dénombré plus de 1200 [Heemstra 87]. Par contre, d'autres auteurs n'en citent que quelques uns. Il est évident que l'estimation du coût d'un logiciel serait une tâche irréalisable s'il fallait tenir compte de plusieurs centaines de facteurs; il est donc nécessaire de se limiter aux plus influants.

Cependant, lorsqu'on consulte la littérature, on peut remarquer des opinions divergentes concernant les facteurs ayant une influence ou pas. Le tableau 1 ci-dessous reprend les différents facteurs dont il est tenu compte dans les modèles d'estimation [Boehm 84]. Les noms des modèles sont placés en abscisse et les facteurs de coût en ordonnée. Les croix indiquent quels sont les facteurs retenus dans chaque modèle. Une parenthèse derrière des croix représente une agrégation de facteurs pour le modèle considéré. (Une brève description de ces modèles peut être trouvée dans [Boehm 81])

FACTOR	SDC. 1965	TRW. 1972	PUTNAM. SLIM	DOTY	RCA. PRICES	IBM	BOEING. 1977	GRC. 1979	COCOMO	SOFCOST	DSN	JENSEN
SOURCE INSTRUCTIONS			x	x		x	x		x	x	x	x
OBJECT INSTRUCTIONS	x	x		x	x							
NUMBER OF ROUTINES	x				x					x		
NUMBER OF DATA ITEMS						x			x	x		
NUMBER OF OUTPUT FORMATS								x			x	
DOCUMENTATION				x		x				x		x
NUMBER OF PERSONNEL			x			x	x			x		x
TYPE	x	x	x	x	x	x	x			x		
COMPLEXITY		x	x		x	x			x	x	x	x
LANGUAGE	x		x				x	x		x	x	
REUSE			x		x		x	x	x	x	x	x
REQUIRED RELIABILITY			x		x				x	x		x
DISPLAY REQUIREMENTS				x						x		x
TIME CONSTRAINT		x	x	x	x	x	x	x	x	x	x	x
STORAGE CONSTRAINT			x	x	x	x		x	x	x	x	x
HARDWARE CONFIGURATION	x				x							
CONCURRENT HARDWARE DEVELOPMENT	x			x	x	x			x	x	x	x
INTERFACING EQUIPMENT, S/W										x	x	
PERSONNEL CAPABILITY			x		x	x			x	x	x	x
PERSONNEL CONTINUITY						x					x	
HARDWARE EXPERIENCE	x		x	x	x	x		x	x	x	x	x
APPLICATIONS EXPERIENCE		x	x		x	x	x	x	x	x	x	x
LANGUAGE EXPERIENCE			x		x	x		x	x	x	x	x
TOOLS AND TECHNIQUES			x		x	x	x		x	x	x	x
CUSTOMER INTERFACE	x					x				x	x	
REQUIREMENTS DEFINITION	x			x		x				x	x	x
REQUIREMENTS VOLATILITY	x			x	x	x		x	x	x	x	x
SCHEDULE			x		x				x	x	x	x
SECURITY						x				x	x	
COMPUTER ACCESS			x	x		x	x		x	x	x	x
TRAVEL/REHOSTING/MULTI-SITE	x			x	x					x	x	x
SUPPORT SOFTWARE MATURITY									x		x	

TABLEAU 1 DIVERS MODELES ET LEURS FACTEURS DE COUT

De ce tableau récapitulatif, nous pouvons retirer d'une part, qu'aucun modèle n'utilise l'entièreté des facteurs proposés, d'autre part, qu'il n'existe pas deux modèles utilisant les mêmes facteurs. De plus, l'influence de chacun des facteurs sur le coût total d'un logiciel diffère d'un modèle à l'autre. Ainsi, un facteur X peut avoir une influence importante dans le modèle A et seulement une faible influence dans le modèle B.

Pour présenter les nombreux facteurs de coût, nous les avons classés en cinq catégories. Nous distinguerons les facteurs en relation avec :

- le développement du logiciel (le quoi)
- les moyens par lesquels le logiciel est développé (les moyens)
- le personnel qui développe le logiciel (par qui)
- les caractéristiques du logiciel (le comment)
- les caractéristiques de l'organisation pour laquelle est développé le logiciel (pour qui).

Cette répartition n'est sûrement pas la seule envisageable, il suffit de consulter [Nanus 64] et [de Kater 87] pour s'en convaincre. Cependant, il nous est paru opportun de choisir celle-ci car elle émane de l'étude la plus complète [Boehm 81] et la plus souvent référencée dans la littérature. Nous l'avons complétée par la cinquième catégorie tirée de [Heemstra 87] et de [Magérat 85]. On pourra trouver dans le tableau 2 un aperçu des facteurs de coût les plus déterminants dans chacune des cinq catégories.

Kostenbepalende factoren m.b.t. het				
Wat (produkt)	*Waarmee* (middelen)	*Door wie* (personeel)	*Hoe* (project)	*Voor wie* (gebruiker)
omvang van de software	beperkingen wat betreft:	kwaliteit	eisen, gesteld aan duur project	participatie
kwaliteit	– executie tijd,	ervaring		aantal gebruikers
omvang van de database	– responsietijd,	verloop personeel	project beheersing	mate waarin specificaties veranderen
complexiteit van de software	– geheugencapaciteit	kwaliteit management		scholing opleiding
documentatie	Tools			
hergebruik	moderne programmeer technieken			

TABLEAU 2 APERÇU DE FACTEURS DE COUT

2.2. Les facteurs liés au logiciel.

2.2.1. La taille du logiciel.

Il existe une relation entre le coût et la taille. Le coût du logiciel à développer augmente en fonction de sa taille, et ceci indépendamment de toute unité de mesure.

Dans la littérature, différentes mesures sont proposées pour calculer la taille d'un logiciel [Craenen 84] :

- le nombre de LOC

- le nombre d'opérateurs et opérandes d'Halstead

- le nombre de fonctionnalités requises par l'utilisateur.

2.2.1.1. Le nombre de LOC

C'est sans aucun doute la mesure la plus critiquée, mais aussi la plus répandue.

Les avantages des LOC sont les suivants : ils sont faciles à mesurer (surtout si ce sont seulement les lignes de code) et il n'existe pas d'ambiguïté sur la mesure. Par contre, les inconvénients d'une telle unité de mesure sont assez évidents : le temps nécessaire pour rédiger une ligne de commentaire ou pour insérer une ligne blanche dans le programme est beaucoup moins important que celui nécessaire à la rédaction d'une ligne de programme. Une taille estimée à 10000 LOC est peu significative en ce qui concerne le temps de développement et le coût. On pourra peut-être améliorer les choses en définissant de manière précise ce qu'il faut entendre par LOC. Cependant, nombreuses sont les divergences quant à la définition du LOC, comme il est illustré dans le tableau 3 [Thibodeau 81].

naam onderzoek	*definiëring omvang software*
aerospace	aantal regels in het programma
Farr en Zagorski	aantal machine instructies
SLIM	aantal statements
Wolverton	aantal machine instructies
Doty	aantal (executeerbare) regels code
PRICE-S	aantal (executeerbare) instructies
Telecote	aantal machine instructies

TABLEAU 3 DEFINITION DE LA TAILLE DANS DIFFERENTES RECHERCHES

Ce manque d'uniformité dans les définitions requiert la prudence lorsque l'on veut estimer la productivité d'un programmeur par cette mesure. En effet, ce dernier pourrait avoir tendance à inclure beaucoup de lignes de commentaire ou encore des lignes blanches afin de faire croire à une augmentation de sa productivité [Conté 86]. De plus, comparer les productivités des programmeurs semble difficile lorsqu'ils utilisent des langages de programmation différents, par exemple APL versus FORTRAN ou COBOL.

2.2.1.2. Le nombre d'opérateurs et opérandes d'Halstead

La méthode d'Halstead ne compte pas le nombre de LOC mais fixe la taille d'un programme en fonction du :

- nombre d'opérateurs uniques (mots réservés)
- nombre d'opérandes uniques (variables et constantes)
- et du nombre de fois que ces opérateurs et opérandes sont utilisés dans le programme.

Ici également, l'imprécision des définitions est présent; il n'existe pas de consensus sur la meilleure façon de classifier et de dénombrer les opérandes et opérateurs. De plus, le résultat (la taille) semble être très dépendant du langage de programmation utilisé. L'avantage de la méthode est le suivant : étant donné un langage de programmation, la dispersion (statistiquement parlant) des estimations de la taille du programme est moins grande que celle des estimations en LOC [Halstead 77] [Heemstra 87].

2.2.1.3. Le nombre de fonctionnalités requises par l'utilisateur

On propose comme alternative au LOC, la quantité de fonctions que doit exécuter le logiciel en termes de données que le programme utilise et génère [Albrecht 79]. Cette méthode est très utilisée pour l'estimation de la taille d'un logiciel, malgré le manque de clarté dans les définitions des fonctions (input, output, inquiry, etc...) et des valeurs à leur attribuer. De plus, la méthode semble mal adaptée aux applications mathématiques, dans lesquelles on trouve peu d'entrées, peu de sorties et peu de fichiers manipulés. Elle est donc plutôt destinée aux applications administratives [Conté 86].

Bien que les chercheurs soient d'accord sur l'existence d'une relation entre la taille et le coût d'un logiciel, leurs avis divergent quant à la forme de cette relation. En général, on pense que les coûts grandissent plus que proportionnellement suite à un accroissement de la taille. Plusieurs études à ce propos montrent que la production d'application suit une loi de rendements décroissants. En d'autres termes, on assiste à une baisse de productivité dès que les applications deviennent trop importantes. Brooks a schématisé cette relation par la figure 2 [Brooks 75].

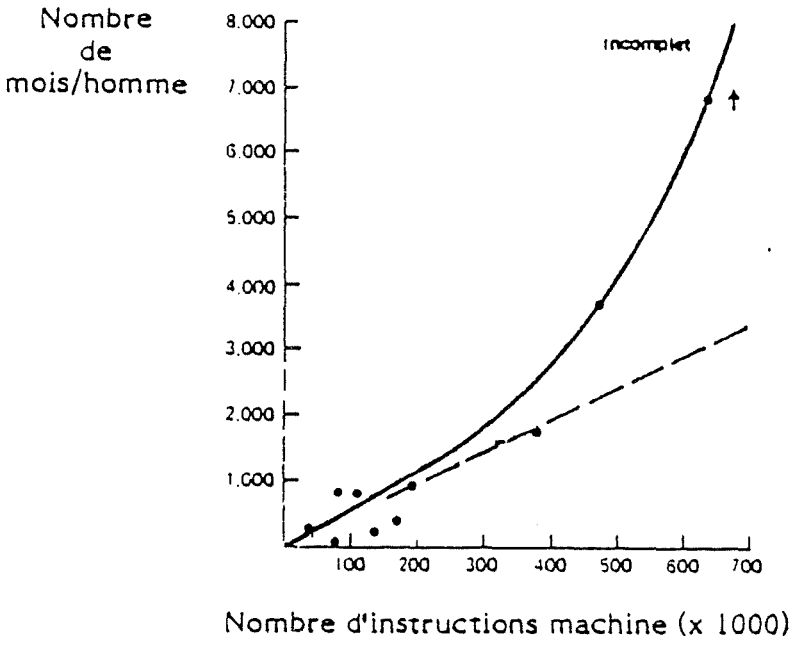


FIGURE 2 EFFORT DE PROGRAMMATION EN FONCTION DE LA TAILLE DES PROGRAMMES

Ce schéma montre clairement que passé un certain seuil la courbe en caractères gras, qui représente la charge réelle, s'écarte de plus en plus de la ligne pointillée représentant la charge théorique; cette dernière est directement proportionnelle au travail à réaliser. L'explication de ce phénomène réside dans le fait que les grands projets présentent par rapport au petits projets une augmentation plus que proportionnelle de la charge, et ceci pour différentes raisons :

- plusieurs services utilisateurs sont concernés, et la difficulté grandit dans la mise au point des cahiers des charges, dans la validation des tests et la coordination du projet

- la complexité naturelle augmente, c'est la conséquence de l'augmentation des liens et des interactions entre les différentes parties du projet

- le personnel affecté augmente, par conséquent les problèmes de découpe du travail et de communication augmentent (cfr. figure 3)

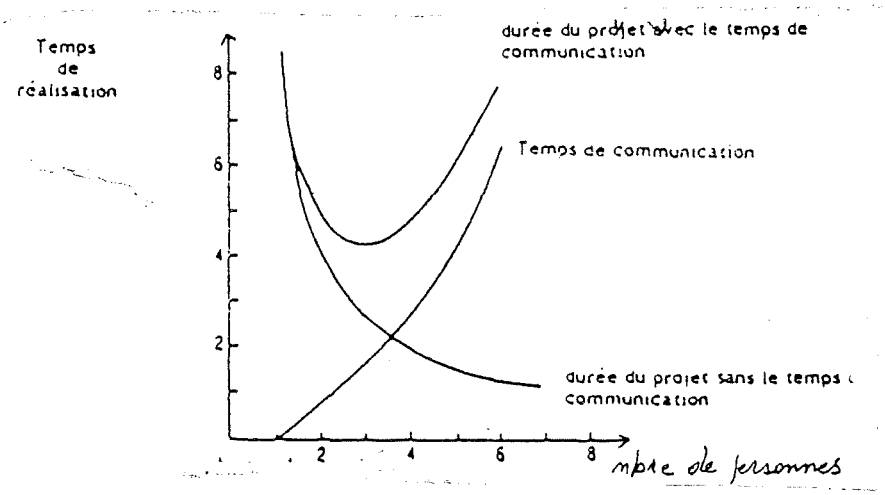


FIGURE 3 LA DUREE DU PROJET AVEC LE TEMPS DE COMMUNICATION

Comme on peut le constater, Surböck fixe l'équipe idéale à trois ou quatre personnes pour être vraiment performante. Cependant, d'autres auteurs ont un avis plus circonstancié; le nombre de personnes dans l'équipe idéale doit être calculé pour chaque projet.

Sur base d'une grande quantité de données de projets, une relation entre la productivité moyenne L (exprimée en LOC par personne et par mois) et la taille moyenne P d'une équipe de développement a été établie dans [Conté 86] :

$$L = 777\ P^{-0.5}$$

En d'autres mots, la productivité baisse à mesure que la taille de l'équipe grandit.

On peut en donner une base théorique issue de l'observation de Brooks concernant le nombre de chemins de communication entre les personnes impliquées dans le développement du projet. Ce nombre est fixé par la taille et la structure de l'équipe. Dans une équipe de P personnes, si chacun des membres doit synchroniser ses activités avec celles des autres, le nombre de chemins de communication est égal à $(P(P-1))/2$. Par exemple, pour une équipe de quatre personnes, cela donne six chemins de communication.

Si chacun des membres ne doit communiquer qu'avec un seul autre, ce nombre se réduit à P. Moins de P chemins de communication ne semble pas raisonnable car dans la pratique, il est rare qu'on ne parle que d'un seul projet à la fois; bien souvent, plusieurs projets sont développés simultanément.

Donc, le nombre de chemins de communication varie de P à environ $P^2/2$. Dans la pratique et dans des organisations hiérarchisées, on peut affirmer que tous les membres d'une équipe ne communiquent pas entre eux; ceci conduit à définir P^α chemins où $1<\alpha<2$.

Pour un seul membre de l'équipe, le nombre de chemins de communication varie entre 1 et P-1.

Si on définit une productivité maximale L et une perte de productivité l par chemin de communication existant, ceci conduit à définir la productivité moyenne de la manière suivante :

$$L_\beta = L - l\ (P - 1)^\beta$$

β est compris dans l'intervalle]0,1] et est une mesure pour le nombre de chemins de communication.

Pour une équipe de P personnes, ceci conduit à la productivité totale

$$P_{tot} = P * L_\beta = P\ (L - l\ (P - 1)^\beta)$$

Dès lors, pour des valeurs fixées de L, l et β, on a une fonction qui, pour des valeurs croissantes de P, croît de 0 à un certain maximum, et ensuite décroît. Il existe donc une taille optimale d'une équipe de développement Popt qui amène une productivité optimale. On pourra trouver dans le tableau 4 ci-après, la productivité obtenue pour différentes tailles de l'équipe. On suppose que la productivité individuelle est de 500 LOC par mois (L = 500) et que la perte de productivité par chemin de communication est de 10 % (l = 50). Il en résulte qu'en cas de complète interaction (β = 1), l'équipe optimale est de 5.5 personnes [Conté 86].

teamgrootte	*individuele produktiviteit*	*totale produktiviteit*
1	500	500
2	450	900
3	400	1200
4	350	1400
5	300	1500
5·5	275	1512
6	250	1500
7	200	1400
8	150	1200

TABLEAU 4 LA PRODUCTIVITE EN FONCTION DE LA TAILLE DE L'EQUIPE

Le résultat obtenu par un tel développement est bien entendu fort théorique, cependant il a l'avantage d'illustrer les rendements décroissants du personnel dans le processus de développement.

Ce point doit retenir l'attention des concepteurs afin que les grands projets soient au maximum découpés en sous- projets de taille moindre (approche top-down). De plus, il faudrait travailler suivant les techniques d'extensions successives,

c'est-à-dire développer la base et puis parallèlement différentes branches qui complètent et affinent cette base (approche bottom-up) [Boehm 81].

2.2.2. La qualité des logiciels.

"Faire de la bonne cuisine demande un certain temps; si on vous fait attendre, c'est pour mieux vous servir et vous plaire" [Magérat 85]. Cette expression illustre très bien le lien existant entre le temps et la qualité.

Il est certain que le développement d'un logiciel requérant une moindre qualité sera réalisé avec moins d'effort qu'un produit de haute qualité. Toutefois, outre les désagréments que la qualité permet d'éviter (perte financière, désagrément au niveau de la clientèle en cas de panne ou d'erreur), la nécessité d'une qualité élevée signifie pour le service de développement une facilité accrue lors de maintenance ultérieure et pour le demandeur lors de l'utilisation du produit.

La qualité d'un logiciel est un facteur de coût très important. Non seulement, il faut tenir compte des exigences fonctionnelles demandées, mais il est nécessaire d'avoir à l'esprit les exigences de performance.

Cependant, il est non seulement difficile de définir la notion de qualité d'un logiciel, mais il est encore plus difficile d'en déterminer l'influence sur le coût total.

W.E. Perry donne dans son livre [Perry 83] un certain nombre de facteurs permettant de cerner la notion de qualité d'un logiciel :

- la précision : la mesure dans laquelle le programme respecte les spécifications et les objectifs de l'utilisateur
- la fiabilité : la mesure dans laquelle le programme exécute ses fonctions avec une précision voulue
- l'efficacité : la quantité de ressources informatiques et de code nécessaires pour exécuter une fonction
- l'intégrité : la mesure dans laquelle l'accès aux données peut être contrôlé pour des personnes non autorisées
- l'utilisabilité : l'effort requis pour apprendre à connaître, fournir, préparer les entrées et à interpréter les sorties du programme
- la maintenabilité : l'effort requis pour localiser et corriger les fautes dans le programme

- la testabilité : l'effort requis pour tester un programme afin d'être certain qu'il réalise les fonctions spécifiées

- la flexibilité : l'effort requis pour apporter des modifications au programme

- la portabilité : l'effort requis pour porter un programme d'un environnement à un autre

- la réutilisabilité : la mesure dans laquelle l'entièreté ou une partie du programme peut être utilisée par d'autres applications

- l'interopérabilité : l'effort requis pour coupler le système à un autre.

Il reste une question primordiale : quelle est l'influence de la qualité d'un logiciel sur son coût total ? Cependant, la littérature reste muette : dans la méthode de Walston et Félix [Walston 77], ce facteur n'est même pas repris. Boehm, non plus, ne pousse pas très loin ses investigations dans ce domaine. Il parle plutôt de fiabilité, ce qui ne constitue, pour Perry, qu'un seul aspect de la qualité.

La fiabilité est définie comme la probabilité qu'un logiciel satisfasse, durant la période d'exécution, aux exigences posées. Boehm ajoute que des tentatives pour quantifier cette probabilité ont échoué et que l'on doit se contenter provisoirement de mesures qualitatives pour l'évaluer. Jusqu'à présent, on ne possède qu'une relation qualitative entre le niveau de fiabilité exigé et le coût total, illustrée dans la figure 4.

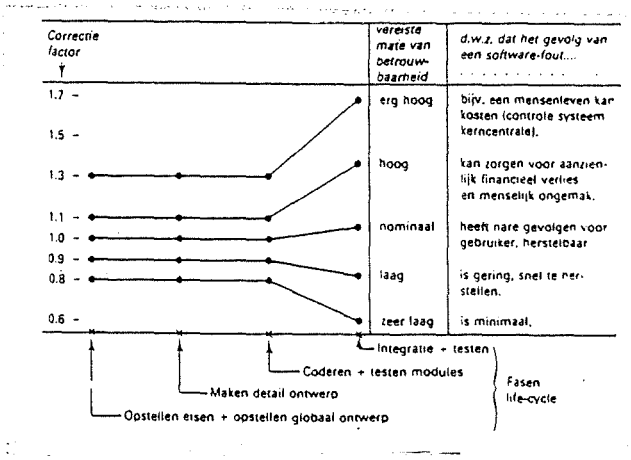


FIGURE 4 INFLUENCE DU FACTEUR FIABILITE SUR LES COUTS

L'échelle qualitative pour évaluer la fiabilité est la suivante : on requiert un niveau très haut, haut, nominal, bas ou très bas de fiabilité.

Suivant le niveau requis de fiabilité et selon la phase du cycle de vie (celui de Boehm 81), on peut déterminer quel sera le facteur de correction : le but de ce dernier est de déterminer l'influence du facteur fiabilité sur le coût total. Ceci se fera en multipliant le coût nominal exprimé en M-H par le facteur de correction.

Une évaluation des recherches dans ce domaine laisse voir qu'il n'y a pas de résultats disponibles qui posent une relation quantitative entre le facteur qualité et les coûts logiciels. Il n'existe qu'une relation qualitative proposée. Les auteurs sont unanimes pour dire que les coûts augmentent lorsqu'un niveau de qualité plus élevé est exigé. Les avis divergent quand il s'agit de définir ce que l'on entend par qualité d'un logiciel, par qualité haute, moyenne ou basse, et par importance de l'augmentation des coûts suite à un accroissement de la qualité requise.

De ces considérations, on peut retenir qu'un niveau plus élevé de qualité exigé a des conséquences sur l'entièreté du cycle de vie d'un logiciel. En effet, des exigences plus importantes concernant la maintenance provoquent un investissement supplémentaire dans les premières phases du cycle de vie, mais conduisent à des économies au niveau de la phase de maintenance [Heemstra 87].

2.2.3. La taille de la base de données.

La complexité et la taille de la base de données (BD) semblent avoir une influence sur les coûts d'un logiciel [Boehm 81]. Cependant, dans de nombreuses méthodes d'estimation, la taille de la BD n'est pas reprise explicitement comme facteur mais est incluse implicitement dans le facteur taille du logiciel. Cette distinction est réalisée dans les méthodes de Boehm et Walston et Félix. Ainsi, on peut trouver dans l'étude de Walston le facteur "nombre de catégories d'items dans la BD par 1000 LOC" pour mesurer la taille de la BD. Boehm, quant à lui, pose une relation entre la taille de la BD et la taille du logiciel. Il s'agit de la quantité de données par rapport à la quantité totale de LOC.

2.2.4. La complexité du logiciel.

Que signifie complexité du logiciel ? Il semble difficile de définir ce concept et de déterminer son influence sur les coûts. Des jugements qualitatifs de la complexité sont souvent donnés dans la littérature pour classer les développements de logiciels dans une classe de complexité, selon les critères suivants [Herrman 84] :

- le nombre d'interactions entre les composants du logiciel,

- le degré de standardisation,

- le domaine d'application du logiciel,

- la difficulté des entrées et sorties.

Du fait de l'aspect qualitatif, un certain degré de subjectivité peut s'introduire dans les jugements. Ceci est vrai pour la définition des différentes classes de complexité, mais aussi pour placer un logiciel à développer dans une de ces classes.

Il existe un manque de critères quantitatifs. Il semble, en effet, que les mesures quantitatives existantes ne sont faites qu'après achèvement du logiciel. Il faut bien avoir à l'esprit qu'il n'est pas requis que la mesure de complexité réalisée après le développement soit identique, à l'unité près, aux estimations de complexité réalisées en début de développement.

Ces estimations réalisées en début de développement sont le plus souvent basées sur des spécifications ou encore, à partir d'exigences formulées par l'utilisateur. Ce dernier éprouve souvent des difficultés pour formuler ses souhaits de manière précise et le développeur, quant à lui, a parfois peu d'expérience dans le domaine d'application concerné. Par là, s'expliquent les écarts entre la complexité attendue et la complexité réelle.

Toutefois, les recherches concernant la complexité sont unanimes : plus grande est la complexité, plus élevés seront les coûts. Cependant, il reste toujours un manque de standardisation dans les définitions de ce facteur.

2.2.5. La quantité de documentation.

La documentation est une partie importante de tout projet. Cependant, le budget qui lui est accordé est trop souvent trop faible, si bien que ceci constitue une des raisons majeures de la sous-estimation des coûts. On oublie trop souvent les coûts relatifs à la documentation.

Par documentation, on entend les spécifications fonctionnelles, le manuel utilisateur, les résultats des tests et les listings de programme [Boehm 81].

L'unité la plus répandue pour mesurer la quantité de documentation est la page. Walston et Félix ont déterminé une relation linéaire entre le nombre le LOC et le nombre de pages de documentation. Cependant, Jones dans [Jones 86] démontra que cette relation n'est pas linéaire : pour des tailles de programmes variant de 1000 à 128000 LOC, la quantité de documentation augmente plus vite que la taille du logiciel. Au dessus du seuil de 128000 LOC, la quantité de documentation augmente moins rapidement et finit par diminuer (cfr figure 5).

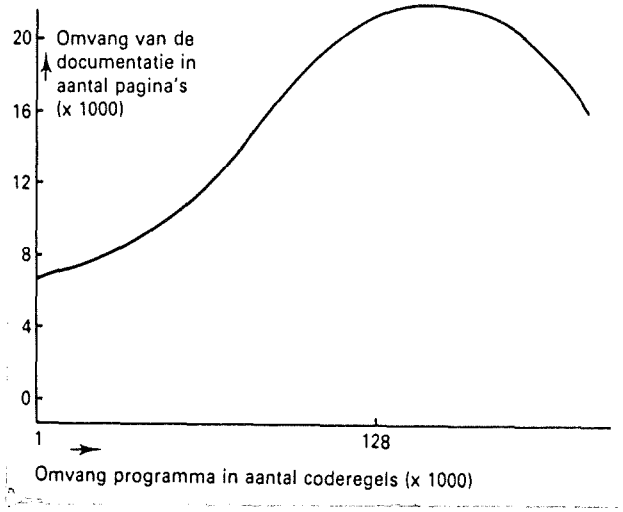


FIGURE 5 LA RELATION ENTRE LA TAILLE D'UN LOGICIEL ET LA QUANTITE DE DOCUMENTATION

On voit qu'il est très difficile de déterminer une relation quantitative entre la taille du logiciel et la quantité de documentation. Il y a un certain nombre de facteurs qui influent sur cette relation : il semble, qu'entre autres, le facteur "type d'application" ait une influence importante sur la quantité de documentation. Cette dernière sera plus importante pour des applications militaires ou encore pour des programmes destinés à la navette spatiale que pour des projets de même taille, mais dans un environnement administratif. Pour les applications militaires, des exigences telles que la fiabilité, l'accessibilité, la sécurité, etc ... auront une influence directe sur la documentation.

A côté des coûts qui sont liés à la production de documentation, il faut envisager les coûts entraînés par le manque de documentation. En effet, peu d'attention est accordée pendant le développement à une bonne documentation. Ceci a pour conséquence de rendre difficile la maintenance et l'utilisation du logiciel. Les recherches dans une documentation imcomplète, voire inexistante, et les réunions exigées par le manque de clarté sont du temps perdu.

2.2.6. Le type d'application.

Une des premières questions que l'on doit se poser lorsqu'on fait une estimation est la suivante : pour quel domaine d'application doit être développé le programme ?

L'expérience montre que le facteur "type d'application" a une influence importante sur les coûts. On peut trouver différentes classifications des logiciels dans la littérature. Nous avons retenu celle proposée dans [Jones 86]. Il réalise une classification basée sur l'objectif pour lequel le programme est développé. Il distingue 11 classes :

- les programmes développés pour une utilisation personnelle

- les programmes développés pour une utilisation au sein d'une organisation

- les programmes développés pour beaucoup d'utilisateurs au sein de la même organisation et en dehors de toute considération de réutilisabilité

- les programmes développés pour des besoins internes à une organisation, mais utilisés à l'extérieur dans une moindre mesure

- les programmes développés pour une utilisation publique

- les programmes que les clients peuvent louer mais pas acheter

- les programmes vendus avec le hardware

- les programmes développés dans un but commercial pour tout public

- les programmes développés dans le cadre d'un contrat

- les programmes développés pour les autorités

- les programmes développés à des fins militaires.

Jones s'étonne que peu de méthodes d'estimation envisagent une classification des types de programmes. Une telle typologie pourrait s'avérer très utile pour la détermination des coûts logiciels. De plus, la majorité des typologies existantes sont basées sur le type de programmation : programmation d'application batch, intelligence artificielle, programmation graphique, programmation non procédurale, etc... Par une classification de ce type, il est bien souvent difficile de classer un programme dans une et une seule classe. Ceci s'oppose à la notion même de classification.

Jones insiste surtout dans sa typologie sur les coûts liés à la documentation, et dans une moindre mesure, sur la capacité de l'équipe de développement. Il conclua en disant que les coûts de documentation augmentent au fur et à mesure que l'on passe de la classe 1 à la classe 11, et que chaque classe a ses caractéristiques propres qui influencent de façon déterminante les coûts.

Walston et Félix réalisent leur classification sur base de la complexité des applications, mais des définitions précises font défaut. Pour une complexité basse, on doit s'attendre à une productivité moyenne de 349 LOC par mois; pour une complexité haute, la productivité moyenne diminue plus que de moitié.

Boehm ne considère pas le facteur "type d'application" comme un facteur à part entière, car il considère qu'il existe des recouvrements avec les facteurs tels que la complexité du logiciel et la taille de la BD.

En conclusion, on peut dire que la plupart des méthodes reconnaissent l'importance du facteur "type d'application" sur l'effort de développement car la productivité du personnel est directement liée au degré de difficulté attaché au type d'application. Cependant, il faut regretter le manque de standardisation dans la typologie d'applications employée dans chaque méthode.

2.2.7. La réutilisabilité.

Au cours du développement d'un logiciel, la question suivante est à poser : quelles sont les parties du logiciel réellement nouvelles et celles que l'on peut reprendre d'anciens logiciels ?

Utiliser les caractéristiques de réutilisabilité de certains éléments de logiciel permet non seulement des économies de coût liées à un développement plus rapide, mais aussi un gain de temps parce que ces éléments ont déjà été testés et parce qu'on connaît leur valeur. Par conséquent, les coûts liés à la maintenance seront réduits.

Cependant, pour réaliser ces économies, des investissements sont nécessaires [Jones 86] :

- il faut instaurer un catalogue dans lequel seront repris tous les composants déjà développés. De plus, les développeurs devront pouvoir facilement discerner quels sont les composants qui sont utilisables ou non (par de bonnes spécifications, par exemple).

- il faut une BD dans laquelle seront stockés les composants.

- une partie du temps de développement sera consacrée à la consultation du catalogue et à la recherche des composants utiles.

- un objectif supplémentaire devra toujours être présent à l'esprit pendant le développement : veiller à la possibilité d'une réutilisation. Cela signifie qu'il faudra constamment adopter une approche modulaire, séparer les composants afin que ceux-ci soient éventuellement réutilisables. Il faudra généraliser la fonction d'un composant quand c'est possible à peu de frais.

- l'intégration d'un composant réutilisé demandera du temps supplémentaire. [Jones 86]

Cependant, les avantages liés à la réutilisabilité surpassent ces coûts supplémentaires. Jones, dans [Jones 86], estime qu'à l'avenir 80 % de chaque logiciel nouveau seront constitués de composants déjà développés. Ceci réduirait le temps de développement de gros projets, de 36 à 8 mois. De plus, une économie de 50 % des coûts serait réalisée.

Alors que Jones considère ce facteur "réutilisabilité" comme l'un des principaux facteurs de coût, il est étonnant de constater que Walston et Félix et même Boehm ne le citent pas. Ceci peut s'expliquer par la récente préoccupation du caractère réutilisable des logiciels.

2.3. Les facteurs liés aux moyens de développement.

Dans cette catégorie de facteurs doivent être inclus :

- les exigences qui concernent le temps d'exécution, le temps de réponse et la capacité mémoire
- l'utilisation de langages de quatrième génération, de générateurs d'application (outils)
- l'utilisation de techniques modernes de programmation [Boehm 81].

2.3.1. Le temps d'exécution, le temps de réponse, la capacité mémoire.

Ces facteurs sont retenus dans tous les modèles d'estimation de coût logiciel. Leur influence peut être schématisée de la façon suivante dans la figure 6 [Heemstra 87].

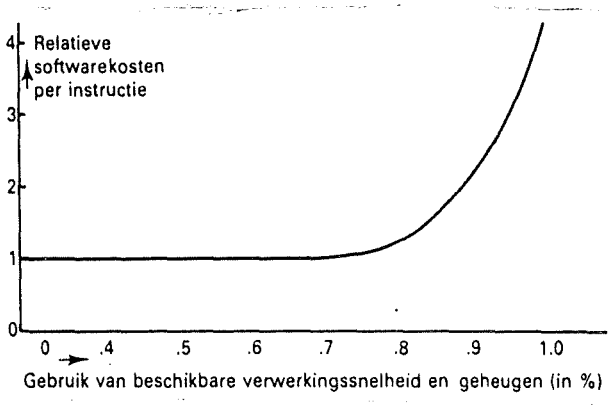


FIGURE 6 INFLUENCE DE LA LIMITE HARDWARE SUR LES COUTS D'UN LOGICIEL

Au fur et à mesure que l'on s'approche de la limite hardware, les coûts s'accroissent rapidement à partir du seuil des 80 % .

Le tableau 5 repris de Boehm illustre les désaccords quant à l'influence de ces facteurs sur les coûts. On y trouve, pour

un certain nombre de modèles, l'influence des exigences en temps d'exécution et l'influence de la capacité mémoire sur la productivité.

naam onderzoek/ model	*invloed van beperkingen in executietijd op de produktiviteit*	*invloed van geheugenbeperkingen op de produktiviteit*
TRW studie [Wol74]	3.00	–
BOEING [Bla77]	3.33 - 6.67	–
DOTY [Her77]	1.33 - 1.77	1.43
GRC [Car79]	1.60	7.00
GTE [Dal79]	1.25	1.25
Walston + Felix [Wal77]	1.37 - 1.77	2.03
COCOMO (Boehm) [Boe81]	1.66	1.56

TABLEAU 5 L'INFLUENCE DES LIMITES EN TEMPS D'EXECUTION ET DE CAPACITE MEMOIRE SUR LA PRODUCTIVITE

Le temps de réponse semble être un facteur déterminant pour les coûts. Par temps de réponse, on peut entendre le temps de réponse à l'utilisation ou le temps de réponse au développement. Dans sa première acceptation, le temps de réponse appartient aux facteurs de coût liés au logiciel; ici, il s'agit plutôt du temps de réponse pendant le développement d'un programme. De rapides temps de réponse par une programmation interactive mènent à des améliorations importantes de productivité durant les phases de codage et de tests [Walston 77].

2.3.2. L'utilisation de moyens d'aide (outils).

Ces moyens d'aide peuvent varier du simple "debugger" aux outils sophistiqués pour l'analyse de l'information et de projets : outils de conception, langages de spécification, systèmes de validation automatique, outils d'aide à la conception de BD, etc... L'utilisation des langages de quatrième génératon pour le développement de logiciel aurait comme conséquence une économie sur les coûts. Une réduction de 90 % du temps de codage et une augmentation de la productivité de l'ordre des 1000 % seraient réalisables [Jones 86]. L'avenir devra nous le prouver. Mais pour le moment, nous pouvons constater que la majorité des outils est utilisée durant la phase de codage qui ne constitue que 20 % du temps total consacré au développement [Heemstra 87]. Une forte réduction du

temps de codage n'a dès lors qu'une influence limitée sur le temps total de développement.

Il existe, jusqu'à présent, peu d'outils pour aider le développeur dans les premières phases du développement; ces phases sont en général très consommatrices en temps.

Beaucoup d'énergie est dépensée dans l'élaboration d'outils d'aide destinés à conduire le développement (le comment). Par contre, les outils qui ont pour but d'aider le chef de projet dans l'exécution de sa tâche sont rares (le quoi), c'est-à-dire des outils de planning, calcul d'échéances et méthodes d'estimation.

On peut espérer qu'une utilisation d'outils d'aide au management de projet, ou d'outils fournissant une aide tout au long du cycle de vie aura comme conséquence une réduction des coûts. Avec le facteur "qualité du personnel", ce facteur conduirait à la plus grosse réduction des coûts de développement [Heemstra 87].

2.3.3. L'utilisation de techniques modernes de programmation.

Walston et Félix distinguent quatre techniques de programmation :

- la programmation structurée
- le contrôle du design et du code (exécution de tests prédéfinis)
- le développement top-down des spécifications des besoins et du code
- l'intégration d'un chef du projet dans l'équipe des programmeurs.

Dans le modèle Cocomo, Boehm y ajoute l'utilisation d'une librairie de programmes et l'utilisation de schémas techniques (diagramme de flux, HIPO). Dans Cocomo, la technique d'intégration d'un chef d'équipe n'est pas reprise car elle fournit des résultats trop aléatoires; avec un très bon chef d'équipe, la productivité peut être très élevée; avec un mauvais, elle peut être très basse. Boehm se distingue encore en donnant simplement l'effet global de l'utilisation des techniques modernes de programmation sur la productivité. Il ne distingue pas l'effet séparé de chacune d'entre elles (cfr tableau 6).

naam onderzoeker	*factoren*	*invloed*
Walston en Felix	gestructureerd programmeren	1.78
	controle van ontwerp en code	1.54
	top-down ontwikkeling	1.64
	inschakeling hoofdprogrammeur-team	1.86
Boehm	gebruik moderne programmeertechnieken	1.49

TABLEAU 6 L'INFLUENCE DES TECHNIQUES DE PROGRMMATION MODERNES SUR LA PRODUCTIVITE

Dans un article plus récent [Boehm 83], Boehm cite l'utilisation des techniques modernes comme un des sept principes de base du génie logiciel. Leurs avantages sont un meilleur aperçu du processus, une détection plus rapide des erreurs, une maintenance plus aisée, etc...

D'une enquête réalisée par les utilisateurs IBM (Guide 79), il ressort qu'il y a un très grand intérêt à utiliser les nouvelles techniques de programmation (cfr tableau 7). En particulier, la méthode des points de fonction proposée par Albrecht avait précisément pour but de mesurer la hausse de productivité suite à l'utilisation de ces techniques [Albrecht 79].

Vraag: welke van de volgende technieken gebruikt U?	*niet*	*in overweging*	*wel*	*totaal*	*% wel*
hoofd programmeur team	134	307	224	665	34
doorlopend testen	51	288	307	646	47.5
top down ontwerp	43	329	332	704	47
gestructureerd programmeren	37	351	412	800	51
HIPO	139	278	188	605	31
programma bibliotheek	109	286	237	632	37
interactief programmeren	86	320	280	686	41
Vraag: als u wel gebruik maakt van een of meerdere van bovengenoemde technieken wat is dan het effect hiervan op	*groot*	*gering*	*geen*	*negatief*	*totaal*
projectbeheer	63	294	206	8	571
communicatie met gebruikers	89	227	252	3	571
stabiliteit organisatie	47	193	303	10	553
nauwkeurigheid van ontwerp	166	297	107	3	573
kwaliteit van de code	206	287	94	2	589
vroeg ontdekken fouten	213	276	87	4	580
produktiviteit	165	350	80	6	601
onderhoudskosten	178	272	108	11	569
gedrag/motivatie ontwikkelaar	108	292	160	20	580

TABLEAU 7 RESULTATS D'UNE ENQUETE IBM SUR L'EMPLOI DES TECHNIQUES DE PROGRAMMATION MODERNES

On peut se demander dans quelle mesure sont utilisées ces nouvelles techniques ? Il semblerait que la programmation structurée, les tests et l'approche top-down soient les techniques les plus courantes (dans 50 % des cas). Lorsque l'on fait usage de ces techniques, il semble que le plus grand effet se porte sur la qualité du code et sur la possibilité de découvrir très rapidement les erreurs. Boehm considère la technique des tests et des validations tout au long du cycle du vie comme un autre des sept principes de base du génie logiciel. Dans ce but, on pourra utiliser entre autres les techniques de prototypage, de simulation, etc...

2.4. Les facteurs liés au personnel.

Il est difficile de définir de manière précise les concepts d'expérience ou de capacité du personnel, et d'en fixer l'influence sur les coûts du logiciel. Dans le tableau 8, on peut trouver un aperçu des facteurs liés au personnel, tels qu'ils sont envisagés dans différents modèles.

Walston en Felix	*COCOMO/ Boehm*	*SPQR/ C.Jones*	*PRICE-S RCA*
algehele ervaring en capaciteit	bekwaamheid analist	ervaring met software ontwikkeling	algehele vaardigheden
ervaring met operationele computer	ervaring met hardware	ervaring met software onderhoud	bekendheid met project
ervaring met programmeertalen	ervaring met program.taal	ervaring met klant	
ervaring met toepassingen van soortgelijke of grotere omvang en complexiteit	ervaring met applicatie bekwaamheid programmeur	persoonlijke (werk)omstandigheden	

TABLEAU 8 LES FACTEURS DEPENDANT DU PERSONNEL DANS DIFFERENTS MODELES

2.4.1. La capacité du personnel.

Dans le modèle de Boehm, il semble que ce facteur a l'influence la plus significative sur les coûts. Par contre, dans le modèle de Walston et Félix, l'expérience du langage de programmation est un des facteurs les plus importants. Jones, quant à lui, ne donne pas de relation quantitative entre les facteurs liés au personnel.

Cependant, malgré l'importance de ce facteur, peu d'études ont été réalisées. Ces études se contentent d'essayer de définir et de mesurer les concepts de capacité et d'expérience du personnel. Elles montrent des relations complexes entre les différents facteurs liés au personnel. Ainsi, la capacité d'un analyste ou d'un programmeur à travailler en collaboration avec d'autres sera dépendante de sa formation, de son expérience, de sa motivation et de ses responsabilités. Cependant, l'influence d'une bonne formation et d'une large expérience peut être complètement anéantie si le travail s'effectue dans des conditions désagréables. Rivalité, ambiance hostile, machination politique sont autant d'obstacles à la bonne réalistion du projet [Heemstra 87]. Jones affirme même que 10 à 15 % des projets échouent pour ces raisons.

2.4.2. L'expérience du personnel.

Toutes les études soulignent l'importance de ce facteur. Heemstra a réalisé la distinction suivante concernant le concept d'expérience [Heemstra 87] :

- l'expérience du type d'application
- l'expérience du langage de programmation
- l'expérience du type de matériel
- l'expérience dans la communication avec l'utilisateur.

Il est nécessaire de nuancer l'affirmation suivante : meilleure est l'expérience, moins élevés seront les coûts.

Le facteur "expérience du type d'application" a sa plus grande influence au début du développement. Une expérience moyenne dans le domaine d'application a pour conséquence, des tests plus importants, des analyses plus longues et moins d'utilisation de code déjà existant. Les recherches fixent une période de cinq années pour passer d'une expérience faible à une expérience bonne.

En ce qui concerne l'expérience du type de langage, on peut dire qu'après une année environ, une bonne expérience est acquise. Cependant, après deux ou trois années d'expérience avec le même langage, on atteint un point de saturation où toute formation supplémentaire semble avoir peu d'effet.

Un tel point de saturation peut également survenir en ce qui concerne l'expérience du type de matériel, de l'operating system, du S.G.B.D, etc... On parlera d'expérience moyenne s'il est question d'une toute nouvelle configuration hardware. En effet, beaucoup de temps sera consacré à la formation du personnel, et cela provoquera beaucoup de retard dans le planning de développement.

L'influence du facteur "expérience dans la communication avec l'utilisateur" est évidente; la différence sera visible si c'est la première ou la dixième fois que le développeur réalise, par exemple, l'informatisation d'une bibliothèque. En effet, la connaissance de l'organisation, la localisation des piéges, et la communication avec l'utilisateur seront d'autant plus rapides que la connaissance du type de l'organisation sera grande.

Des résultats quantitatifs pour confirmer ces affirmations sont difficilement disponibles. Cependant, le tableau 9 montre l'influence de la formation et de l'expérience sur la productivité. Cette enquête a été réalisée par Jeffery et Lawrence de façon empirique [Jeffery 85]. Elle montre que la relation entre l'expérience et la productivité est ambiguë et peu stable. En effet, sur base de ces chiffres, il n'est pas évident qu'une meilleure formation ou qu'une meilleure expérience conduise à un accroissement de productivité.

Opleiding	*jaren ervaring met COBOL*	*gemiddelde*	
lagere technische opleiding of diploma inleiding programmeren/informatica	1		2.1
	4		7.5
		gemiddeld	3.9
voltooide middelbare schoolopleiding	1		8.7
	2		8.5
	3		6.8
	4 of meer		12.3
		gemiddeld	10.1
voltooide hogere beroepsopleiding	1		8.9
	2		7.1
	3		9.7
	4 of meer		6.3
		gemiddeld	8.1
voltooide universitaire opleiding computerkunde/informatica	1		8.6
	2		14.8
	4 of meer		5.1
		gemiddeld	10.6

TABLEAU 9 L'INFLUENCE DE LA FORMATION ET DE L'EXPERIENCE SUR LA PRODUCTIVITE

2.4.3. La baisse du nombre de spécialistes.

Un facteur qui a aussi son importance est la baisse du nombre de spécialistes durant la réalisation du logiciel. Dans la plupart des cas, le planning devra être adapté de manière drastique parce que un ou plusieurs des développeurs (difficilement remplaçables) ont abandonné le projet à mi-chemin. Il est cependant difficile de déterminer quel en sera l'effet sur les coûts d'un logiciel. En règle générale, un pourcentage du coût total sera ajouté [Heemstra 87].

Malgré les problèmes rencontrés pour définir de façon précise les facteurs liés au personnel, pour les mesurer de manière objective et pour en déterminer les effets exacts sur les coûts du logiciel, ces facteurs sont considérés dans la plupart des méthodes d'estimation comme ayant une influence dominante.

Devant l'intérêt d'un tel facteur, Boehm dans [Boehm 83], recommande d'utiliser mieux et moins de personnel. Ceci signifie :

- qu'il ne faut pas essayer d'ajouter du personnel afin de combler un retard, car plus de personnel implique un accroissement de temps consacré à la communication. "Adding manpower to a late project, makes it later" [Brooks 75], c'est la notion de "non-interchangeabilité" entre le personnel et le temps

- qu'il ne faut pas mettre trop de personnes au début du projet

- qu'il faut proposer un plan de carrière aux développeurs et bien rémunérer les bonnes prestations

- qu'il ne faut pas laisser participer au développement les mauvais éléments

- qu'il faut faire usage d'outils automatisés par lesquels moins de personnel sera nécessaire, et dès lors, moins de temps sera consacré aux communications.

2.5. Les facteurs liés au logiciel.

Cette catégorie de facteurs de coût comprend :

- les exigences posées sur la durée de développement

- la gestion du projet.

2.5.1. Les exigences posées sur la durée de développement.

Quelles sont les conséquences sur les coûts d'un logiciel, si un projet doit être plus rapidement terminé que prévu ou s'il demande plus de temps que prévu ?

Jones cite quatre raisons pour lesquelles les plannings sont souvent optimistes [Jones 86] :

- des considérations commerciales ("Price to win" [Boehm 81]). En vue de remporter un marché, on fixe l'échéance de la livraison du logiciel, de façon trop optimiste.

- au fur et à mesure que l'on progresse dans le développement, un grand nombre de projets s'avèrent de plus grande taille que prévu. Les spécifications ne semblent plus être correctes et/ou complètes; cependant, le fournisseur du logiciel est tenu par une date de livraison fixée de manière contractuelle.

- la loi de Parkinson : "work expands to fill the available volume" [Parkinson 57]. Si l'échéance prévue pour le temps de développement d'un logiciel dépasse le temps qui est réellement nécessaire, alors la loi de Parkinson affirme que le personnel utilisera ce temps excédentaire pour des formations complémentaires, pour des activités privées, etc... de telle façon que le temps réel passé au

développement du projet et son coût seront égaux aux estimations.

- des fautes d'estimation : étant donné que les estimations sont le plus souvent réalisées sur base de jugements intuitifs, la probabilité que les échéances ne soient pas respectées est grande.

Dans le modèle Cocomo, Boehm tient compte dans ses coûts, de l'influence d'une compression de la durée de développement ou d'un retard par rapport au planning. Des facteurs tels que les heures supplémentaires, travailler durant le week-end, travailler sous des contraintes d'échéance, travailler avec le chef de projet "dans le dos", le stress, etc... sont envisagés. Cependant, dans le modèle Cocomo, l'influence de ces facteurs semble être moyenne : une compression de 75 % du temps prévu n'occasionne qu'une augmentation des coûts de 23 % . L'influence d'un retard semble être encore plus faible : une durée de développement dépassant de 60 % le temps prévu n'engendre qu'un accroissement des coûts de 10 % [Boehm 81].

Dans le modèle de Putnam [Putnam 78], l'équation concernant la prévision et la variation de l'effort de développement, suite à une compression ou à un allongement du temps de développement, provoquent respectivement des pénalités ou des réductions de l'effort (plus fortes que dans le modèle de Boehm) :

$$EFFORT = C / T_d^4 \quad \text{où} \quad T_d = \text{temps de développement}$$

Par exemple, doubler le temps de développement d'un projet estimé à 100 M-H devrait réduire, selon l'équation, l'effort nécessaire de $(100/2^4)$ M-H, c'est-à-dire de 6,25 M-H.

On remarquera cependant que d'autres études, en particulier celle de Walston et Félix, ne tiennent pas compte de ce facteur.

2.5.2. La gestion du projet.

On doit tenir compte ici de toutes les activités administratives inhérentes au développement d'un logiciel, des coûts liés à l'utilisation de locaux, des frais de voyage, de formation, de gestion, etc... Dans la plupart des cas, ces frais ne sont pas retenus comme des coûts liés au développement. Cependant, des situations où l'utilisateur final est fort éloigné du centre de développement entraînent des coûts de voyage qui sont non négligeables dans les premières et dernières phases du développement.

2.6. Les facteurs liés aux utilisateurs.

Cette catégorie de facteurs concerne l'influence qu'a l'utilisateur sur les coûts de développement. Heemstra cite les facteurs suivants :

- la mesure dans laquelle l'utilisateur est impliqué dans le développement
- le nombre d'utilisateurs différents
- en interprétant de manière large le concept de développement de logiciel, on pourra tenir compte des coûts liés à la formation, au recyclage, à l'installation du logiciel, etc...
- les coûts liés à l'aspect dynamique d'une organisation, c'est-à-dire savoir si, une fois les spécifications établies, celles-ci seront stables. Le modèle de Boehm ne prend pas ces facteurs en considération. Par contre, Walston et Félix distinguent ces facteurs :
  - la complexité de l'interface utilisateur
  - la participation de l'utilisateur dans la détermination des exigences
  - l'expérience de l'utilisateur dans le domaine de l'application.

Les deux premiers facteurs semblent avoir dans cette étude une influence dominante : la productivité moyenne tombe de 491 LOC par mois à 205 lorsque la participation de l'utilisateur à la formulation des exigences passe de "peu" à "beaucoup". Lorsque l'on parle d'une complexité importante d'un interface utilisateur, la productivité moyenne des développeurs se situe autour de 124 LOC contre 500 lorsque l'interface est de complexité faible [Walston 77].

L'implication de l'utilisateur, en particulier dans les premières phases du développement, est une condition déterminante pour obtenir un produit conforme. Cependant, cette efficacité sera dépendante de l'expérience du concepteur avec le type d'application, de l'expérience de l'utilisateur avec l'informatisation, etc...

Trop souvent, on se rend seulement compte à la livraison du logiciel que celui-ci ne satisfait pas aux désirs de l'utilisateur. Les activités importantes de maintenance sont alors nécessaires pour le rendre conforme. Il n'est dès lors plus question d'économie de coût, car de telles opérations prennent en général énormément de temps et créent des retards dans le planning. Ce sont là des explications à ce que l'on entend trop souvent dire : 70 % du cycle de vie d'un logiciel sont consacrés à la phase de maintenance [Bemelmans 81] [Martin 83].

De plus, lorsque le même logiciel est développé pour différents utilisateurs, le temps de développement et les coûts s'accroissent. La recherche du "plus grand commun diviseur" dans les spécifications, le fait de contrôler si l'exigence X du client A a la même signification que l'exigence Y du client B, augmentent considérablement la complexité du processus de développement. Dans la pratique, on utilise souvent la règle suivante : multiplier par deux le temps de développement lorsqu'il y a deux utilisateurs [Heemstra 87]. Cependant, cette règle n'a pas de fondement théorique.

Le temps de développement peut être encore allonger si les spécifications du logiciel sont très souvent modifiées. Boehm a fait une étude dans ce domaine : il parle d'un accroissement des coûts de l'ordre d'un facteur quatre lorsque le logiciel est soumis à d'incessantes modifications de spécifications. Il est recommandé dans ce cas de recourir à la technique du prototypage.

3. Conclusion.

Dans le processus d'identification et d'analyse des facteurs de coût, on peut rencontrer un certain nombre de problèmes que reprennent Nanus et Heemstra [Nanus 64] [Heemstra 87].

3.1. Le manque de consensus dans la terminologie utilisée.

Il n'existe pas de reconnaissance universelle des définitions des termes utilisés dans le processus de programmation. Ainsi, par exemple, les mots "debugging", "test" et "validation de programme" peuvent décrire le même processus; un programmeur dans une organisation peut être appelé encodeur dans une autre, ou encore analyste dans une troisième. Il n'y a pas de définition précise pour des concepts tels que "taille du logiciel", "qualité du logiciel", "expérience", "capacité du personnel", etc...

3.2. Une médiocre définition de la qualité d'un logiciel.

Apparemment, il y a peu de consensus concernant la définition de ces attributs qui caractérisent la nature ou la qualité d'un programme. Par exemple, on entend les programmeurs parler en termes de "flexibilité", "d'économie de place

mémoire" et de "maintenabilité", mais il semble qu'il n'existe pas d'accord sur les critères utilisés pour la comparaison de programmes sur base de ces attributs.

3.3. La pauvre qualité des données concernant les coûts.

Les méthodes actuelles en vue de récolter des données sur les coûts semblent plutôt orientées à des fins comptables plus que pour une utilisation orientée planning ou contrôle. Par exemple, les données concernant les coûts sont habituellement regroupées par unité organisationnelle plus que par produit ou fonction à réaliser.

3.4. La nature dynamique du domaine.

On ne peut pas encore à proprement parler de stabilité dans les techniques de programmation : il existe une grande variété de techniques et d'approches qui sont soit utilisées, soit en cours de développement. Le processus de programmation "se cherche" encore. Par conséquent, toute étude des facteurs de coût doit prendre en considération aussi bien l'historique que les tendances de la technologie de programmation.

3.5. L'aspect non quantitatif de certains facteurs.

L'expérience a montré que beaucoup de facteurs affectant le coût d'un programme sont, par nature, qualitatifs. Dans certains cas, il est possible de prédire au moins la direction dans laquelle le coût sera influencé par une augmentation d'un certain facteur. Par exemple, on peut s'attendre à ce qu'une meilleure expérience dans un type particulier d'application produise comme effet une diminution du coût de réalisation du logiciel. Dans d'autres cas, des facteurs qualitatifs apparaissent comme ayant un effet non monotonique sur les coûts lorsqu'un certain facteur varie : il y a d'abord une augmentation et ensuite une diminution du coût total. Bien évidemment, l'ampleur de l'effet des facteurs qualitatifs sur les coûts est plus difficile à déterminer que la direction de cet effet (à la hausse ou à la baisse). En général, on utilisera des unités de mesure telles que "beaucoup", "moyen" et "peu".

3.6. L'objectivité.

Pour tous ces facteurs où des problèmes de quantification surviennent, il existe un danger de subjectivité. Par exemple, ce qu'un développeur considère comme faisant partie de la catégorie "beaucoup", peut être vu par un autre développeur comme faisant partie de la catégorie "moyen".

3.7. La corrélation.

Il serait naïf de croire que chaque facteur de coût est indépendant. Il se peut, qu'à première vue, un facteur exerce à lui seul peu d'influence sur les coûts, mais du fait de son interdépendance avec un autre facteur, il influe dans une importante mesure sur les coûts du logiciel.

3.8. L'estimation.

L'incertitude sur les valeurs des facteurs de coût joue un rôle primordial sur la justesse de l'estimation des coûts et de la durée de développement d'un logiciel.

CHAPITRE 3.

QUAND EST-IL OPPORTUN DANS LE CYCLE DE VIE D'UN PROJET DE RECOURIR A DES ESTIMATIONS ?

1. Introduction.

Nous voudrions, d'une part, donner une définition de ce que nous entendons par cycle de vie d'un projet, d'autre part, déterminer le moment idéal pour procéder à des estimations du coût d'un projet en fonction des informations disponibles à ce moment. Nous définirons également ce qu'il faut entendre par estimation à priori et estimation à postériori.

Selon R.Fairley, planifier le processus de développement d'un logiciel inclut la définition d'un modèle du cycle de vie. Ce modèle comprend toutes les activités requises pour définir, développer, tester, livrer, mettre en production et maintenir un logiciel. Différents modèles mettent en exergue plusieurs aspects du cycle de vie, mais aucun de ces modèles n'est approprié pour tous les logiciels. Il est donc important de définir un modèle de cycle de vie pour chaque projet parce qu'il fournit une base pour classifier et contrôler les différentes activités nécessaires à son développement et sa maintenance. Ce modèle, s'il est accepté et compris par toutes les parties impliquées dans le développement, améliore la communication et permet une meilleure gestion des ressources, un meilleur contrôle des coûts et une meilleure qualité du produit [Fairley 87].

Etant donnés ces considérations et le nombre de cycles de vie définis dans la littérature, nous avons choisi de présenter celui qui est suivi à la C.G.E.R. pour le développement de ses applications de type gestion. Désormais, dans la suite de notre travail, toute allusion à un modèle de cycle de vie concernera celui de la C.G.E.R..

2. Description simplifiée du modèle de cycle de vie défini à la C.G.E.R.

Cette description est tirée du mémoire de mesdemoiselles Delieux et Delvaux [Delieux-Delvaux 87]. Ce modèle est composé d'un certain nombre de phases bien définies :

- la définition de projet,
- l'analyse conceptuelle,
- l'analyse fonctionnelle,
- l'analyse technique,
- la programmation,
- les tests,

- le passage en production,

- la maintenance.

2.1. La définition de projet.

La définition de projet précise les frontières, les contraintes, les objectifs et les besoins relatifs au projet à développer, et ce en fonction du système existant. Cette étape comporte trois volets : l'étude de l'existant, les objectifs et contraintes du nouveau système et la liste des solutions possibles. Une fois cette étape réalisée, une instance de décision disposera de toutes les informations nécessaires sur lesquelles elle se basera pour effectuer un choix parmi les différentes solutions proposées.

2.1.1. L'étude de l'existant.

Elle résulte d'interviews et d'études de documents réalisées par les membres du groupe de travail assignés au projet. Elle comporte trois volets : Le recensement des activités et tâches de l'existant, celui des documents et fichiers actuels ainsi que la représentation schématique du système existant.

Suite à cette analyse, il sera procédé à la critique du système existant où l'on dégagera les anomalies, les dysfonctionnements, etc...

2.1.2. Les objectifs et contraintes du nouveau système.

En ce qui concerne les objectifs, on distingue les objectifs stratégiques et les besoins exprimés par l'utilisateur.

Les objectifs doivent répondre aux anomalies du système existant; si tel n'est pas le cas, une explication est souhaitée.

Les contraintes sont temporelles, financières, légales, réglementaires ou définies en fonction de l'environnement (sécurité, temps de réponse, communication avec d'autres applications, etc...).

2.1.3. Les solutions possibles.

Il est souhaitable d'envisager plusieurs solutions possibles à soumettre à une instance de décision. Pour chacune de ces solutions, on fournira une description précise, un modèle, une analyse des coûts ainsi qu'un planning recouvrant

toutes les phases de développement jusqu'au passage en production.

2.2. L'analyse conceptuelle.

L'analyse conceptuelle consiste en l'élaboration de modèles du réel perçu, modèles qui respectent une contrainte d'invariance. En effet, ils expriment la sémantique de la structuration des informations à l'exclusion de toute contingence technique ou organisationnelle. Ces modèles exigent de la part de leur auteur qu'il se place à un niveau d'abstraction suffisant pour rester indépendant de tout choix d'implémentation.

L'analyse conceptuelle comporte deux volets : l'analyse des données et l'analyse des traitements, analyses qui ne sont pas indépendantes.

2.2.1. L'analyse conceptuelle des données.

A l'issue de cette analyse, un modèle conceptuel des données sera fourni. Celui-ci, s'il est approuvé, constituera une référence pour la construction des supports d'information dans les étapes ultérieures. L'approche qui a été choisie à la C.G.E.R. est celle du modèle entité/relation, étant donné que ce modèle est très répandu et qu'il est basé sur un petit nombre de concepts précis qui offrent une grande souplesse d'adaptation aux cas réels. Il constitue un excellent moyen d'aborder un problème, ainsi qu'un outil privilégié de dialogue entre utilisateurs et informaticiens.

2.2.2. L'analyse conceptuelle des traitements.

Cette analyse fait appel à plusieurs modèles : un modèle de découpe que nous appellerons modèle de décomposition des traitements, un modèle de description des traitements et un modèle de la dynamique.

2.2.2.1. Le modèle de décomposition des traitements.

Il constitue une approche top-down du problème, c'est-à-dire que l'on réduit un problème global en une série de sous-problèmes moins complexes. Le résultat obtenu est une arborescence où tout élément de niveau intermédiaire i (i>1) provient de la décomposition d'un seul élément de niveau i-1 et se décompose en n (n>=1) éléments de niveau i+1. Chaque élément correspondra à un traitement bien déterminé et la sémantique associée à ce modèle est qu'un traitement de niveau i+1 "fait partie" d'un traitement de niveau i [Bodart 83].

2.2.2.2. Le modèle de description d'un élément de la décomposition.

Une fois la découpe des traitements achevée, les éléments de la décomposition peuvent être décrits via un tableau HIPO (Hierarchical Input Process Output). Ce tableau se présente comme suit :

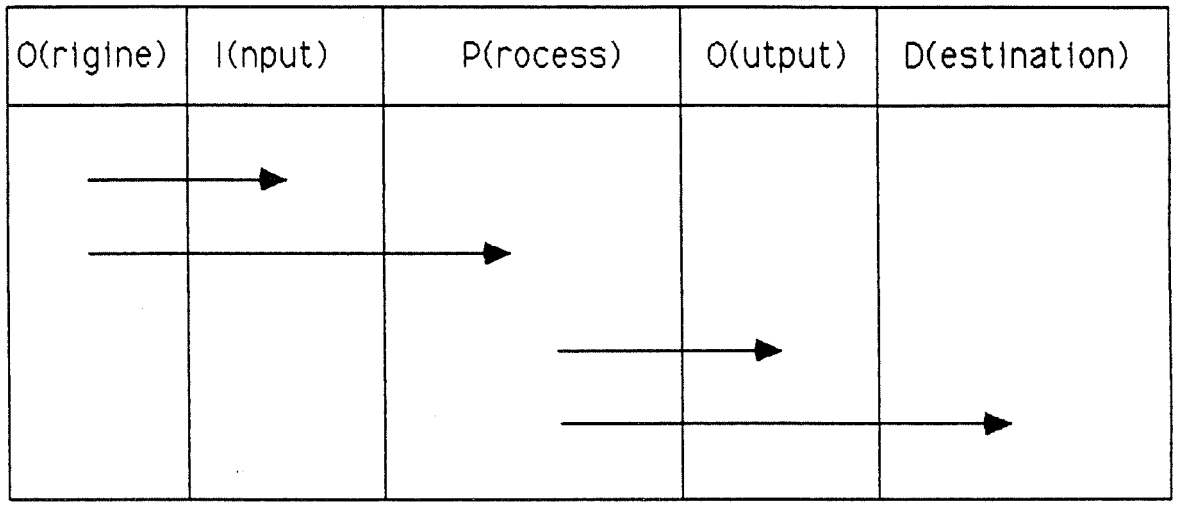


TABLEAU 10 LE TABLEAU HIPO

- la colonne I(nput) contient le nom des objets utilisés en entrée par l'élément de la décomposition. Ces objets sont soit des entités (au sens du modèle entité/relation), soit des informations circulant d'un élément de la décomposition à un autre, ayant souvent existence temporaire et pour lesquelles aucune décision de matérialisation n'a été prise.

- la colonne O(rigine) reprend l'origine de chacun de ces objets.

- la colonne O(utput) renferme les objets qui sont produits ou traités par l'élément de la décomposition.

- la colonne D(estination) indique la destination de ces objets.

- la partie P(rocess) reprend une description du traitement associé à l'élément de la décomposition.

Ce tableau sera accompagné d'un texte précisant l'objectif de cet élément de décomposition, les évènements initiateurs et induits, plus éventuellement quelques remarques.

2.2.2.3. Le modèle dynamique.

Ce modèle a été introduit pour servir de complément au modèle de décomposition des traitements quand la découpe résultant de ce dernier est considérée comme étant trop stricte pour répondre à certains besoins.

Cette définition est délibérément floue car ce modèle n'est pas utile pour la suite du travail.

2.3. L'analyse fonctionnelle

"Cette phase a pour but de fournir l'information nécessaire à la construction de l'architecture des programmes et de fournir au responsable de la BD un modèle logique des données qui lui permettra de construire la BD physique. A l'issue de cette phase, aura été précisée de façon claire et complète la manière dont l'utilisateur veut travailler avec son système" [Mai **].

Le travail effectué en vue d'atteindre ces objectifs se base sur les différents modèles élaborés dans l'analyse conceptuelle : le modèle conceptuel des données, le modèle de décomposition des traitements.

De la même manière que pour l'analyse conceptuelle, on distinguera de nouveau l'analyse des données et l'analyse des traitements, quoique leur interaction soit plus importante ici.

La spécification des caractéristiques fonctionnelles introduites à partir de la phase de l'analyse conceptuelle ne sera pas négligée.

2.3.1. L'analyse fonctionnelle des traitements.

Cette analyse se déroule à deux niveaux : celui des traitements interactifs et celui des traitements batch.

En ce qui concerne les traitements interactifs, l'analyse fonctionnelle consiste à décrire pour chacun d'entre eux le dialogue utilisateur/système, c'est-à-dire un scénario élaboré sur base d'actions effectuées d'une part, par l'utilisateur et, d'autre part, par le traitement (côté système); on procédera ensuite à la spécification des écrans supportant le dialogue.

"En ce qui concerne les tâches batch, on définit la logique de l'enchaînement des traitements ainsi que les états successifs" [Mai **].

"Après l'analyse, on associera à chaque traitement interactif ou batch des paramètres qualitatifs et quantitatifs (on line ou batch, priorité, périodicité, temps de réponse, etc...)" [Graas **].

2.3.2. L'analyse fonctionnelle des données.

Le but de l'analyse fonctionnelle des données est de transformer le modèle conceptuel des données en un modèle logique des données qui tient compte des accès et qui a subi le processus de normalisation.

2.4. L'analyse technique.

Le modèle logique élaboré lors de l'analyse fonctionnelle est traduit en un modèle physique bien structuré, performant et respectant les contraintes du SGBD utilisé.

Ce travail est effectué par les auteurs du modèle des données en collaboration avec le groupe "data base administration" et tiendra compte des volumes et des fréquences d'accès dégagés dans la phase précédente.

Du côté des traitements, les analystes procéderont à une description de la structure et du mode d'enchaînement des applications et des programmes.

2.5. La programmation.

A la C.G.E.R., la méthode de Jackson est recommandée.

2.6. Les tests.

Les programmeurs sont responsables de la correction de leurs programmes et du déroulement des tests d'intégration.

Les utilisateurs procèdent aux tests du produit en vue de découvrir si ce dernier répond bien à leurs désidérata. Les spécifications initiales et les structures de données jouent un rôle très important à ce niveau. En effet, les utilisateurs doivent les étudier pour :

- définir les procédures de test,

- créer l'environnement nécessaire à l'exécution de tests significatifs et en particulier générer les données de tests,

- prévoir les résultats que devra donner chacun des tests envisagés,

- exécuter les tests [Mai **].

Rappelons qu'à ce niveau, intervient l'ensemble des caractéristiques fonctionnelles définies tout au long de l'analyse.

2.7. Le passage en production.

Lors du développement, le programme se trouve dans une librairie de développement; le passage en production consiste à transférer le programme dans une librairie de production avec un minimum de documentation pour permettre à l'opérateur de lancer le nouveau système (le nom des fichiers, leurs caractéristiques, etc...).

2.8. La maintenance.

Jusqu'à présent, nous avons brièvement défini les sept étapes du développement d'un projet dont il faudra estimer la durée et le coût. La huitième étape, c'est-à-dire la phase de maintenance n'entre pas en jeu dans notre estimation.

3. Le moment idéal pour estimer la durée et le coût de développement d'un logiciel.

Selon nous, le moment idéal pour estimer la durée et le coût d'un logiciel se situe en fin de définition de projet. En effet, c'est au moment où il faut faire un choix entre des solutions alternatives qu'il serait intéressant de les estimer. Malheureusement, à ce moment, les informations dont on dispose au sujet des traitements et des données sont assez maigres; l'estimation en sera d'autant moins précise.

C'est ici qu'il faut insister sur le dilemme entre la précision de l'estimation et la phase du cycle de vie dans laquelle l'estimation doit être effectuée. Dans [Boehm 81], ce dilemme est bien schématisé par la figure 7.

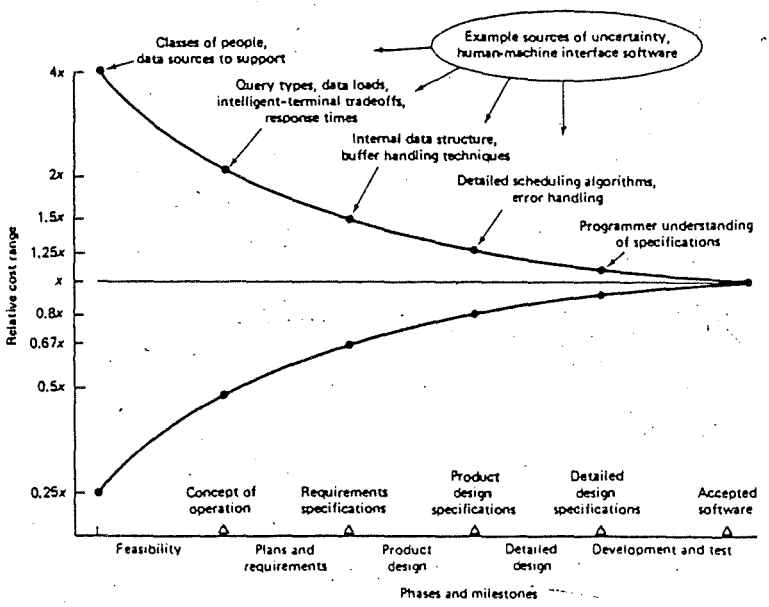


FIGURE 7 PRECISION DE L'ESTIMATION EN FONCTION DE LA PHASE DE DEVELOPPEMENT

Cette figure montre tout simplement qu'une estimation est plus précise si elle est réalisée plus tard dans le cycle de vie d'un projet car l'information disponible concernant le projet est plus précise et abondante.

Un autre moment tout aussi acceptable pour réaliser une estimation est la fin de l'analyse conceptuelle car on dispose des mêmes informations qu'en fin de développement mais à un niveau conceptuel ou logique.

Nous qualifierons d'estimation à priori, toute estimation réalisée en fin de définition de projet ou encore en fin d'analyse conceptuelle.

Une estimation réalisée plus tard dans le cycle de vie perd, à notre avis, son côté prévisionnel et devient plutôt une mesure de productivité pouvant être utilisée dans un but de contrôle. Une telle estimation, si elle est réalisée après la phase de programmation, est une estimation à postériori.

CHAPITRE 4.

QUEL EST L'ETAT DE L'ART DANS LE DOMAINE DES METHODES D'ESTIMATION ?

1. Introduction.

Depuis des années, l'estimation de la taille , du temps de développement et du coût d'un logiciel fut un processus intuitif; chacun se basait sur sa propre expérience et sur des normes industrielles existantes. Ces méthodes semblent fonctionner convenablement pour de petits et relativement simples systèmes. Cependant, si le système est gros et compliqué, l'intuition semble défaillir et il faut alors procéder à des approches quantitatives des paramètres influant sur l'estimation.

Depuis 1965, plusieurs modèles ont adopté cette approche quantitative; aujourd'hui, il en existe plus d'une vingtaine. Cependant, nous nous limiterons à tracer l'évolution historique du processus d'estimation des coûts en décrivant la philosophie générale des modèles d'estimation les plus abondamment traités dans la littérature.

2. Les méthodes de conception de modèles d'estimation.

Nous avons constaté que les modèles d'estimation ont été créés selon un des deux types d'approche suivants : par l'expérimentation en laboratoire ou sur base d'analyses de données de projets réels déjà terminés.

2.1. L'expérimentation.

Dans de telles expériences, pour fixer l'influence d'un facteur de coût, on le fait varier tandis que les autres sont maintenus constants. Afin d'illustrer cette technique, on peut s'interroger sur l'influence de la documentation lors de la maintenance. Par exemple, on peut demander à deux groupes de programmeurs de répondre à un certain nombre de questions au sujet d'un et un seul texte de programme. Un groupe reçoit le texte sans commentaires et l'autre groupe, avec des commentaires. A la lumière des résultats obtenus, on peut valider l'hypothèse donnée, à savoir que les commentaires ont un effet positif sur la rapidité de compréhension d'un texte de programme [van Vliet 87].

Ce type d'expérimentation est généralement pratiqué dans des laboratoires universitaires avec la collaboration des étudiants. Il n'est pas évident que les résultats obtenus soient valables dans la pratique industrielle car ces deux environnements sont difficilement comparables. De plus, dans la

pratique, il existe de nombreuses interrelations très complexes entre les différents facteurs de coût [van Vliet 87].

Dans tous les cas, des adaptations sont nécessaires pour intégrer ces modèles expérimentaux dans un environnement industriel.

2.2. L'analyse des données de projets terminés.

La seconde façon d'élaborer un modèle est de se baser sur les données de projets réalisés antérieurement. Ainsi, une entreprise peut rassembler les données suivantes : le temps consacré aux diverses phases de développement, la qualification du personnel, les corrections d'erreurs durant les tests et l'installation, la complexité, la fiabilité et d'autres caractéristiques liées au logiciel, la taille du code, etc...

Sur base d'une analyse (statistique) de ces données, on peut arriver à l'expression d'une relation entre différentes grandeurs, c'est-à-dire une relation entre l'effort et la taille.

L'utilisation et la fiabilité de ces expressions sont naturellement très dépendantes de la fiabilité des données sur lesquelles elles sont basées. La façon selon laquelle on est arrivé à de telles expressions quantitatives conduit encore à d'autres limitations en ce qui concerne l'utilisation des modèles basés sur ces expressions. En effet, l'utilisation d'un de ces modèles pour estimer un nouveau projet n'est possible que si les données de ce nouveau projet sont comparables avec celles sur lesquelles se base le modèle. Les données qui ont été rassemblées pour un type de projet dans un type d'organisation ne peuvent être utilisées sans adaptations pour un projet d'un autre type dans une autre organisation. C'est la raison pour laquelle les modèles conçus par Boehm et, Walston et Félix, peuvent fournir des résultats fort différents pour le même problème.

Il existe d'autres raisons pour expliquer les divergences dans les résultats de différents modèles :

- les unités de mesure employées dans les modèles sont définies de manière différente,

- dans le concept d'effort, les mêmes activités ne sont pas prises en considération. Dans certains cas, ce sont les activités qui vont de la phase de définition du projet à la phase du passage en production; dans d'autres cas, l'effort inclut les activités de maintenance.

3. Présentation d'une méthode d'estimation pour de petits projets (la méthode d'Halstead).

Nous avons constaté que les premières expériences de mesure de l'effort ont été réalisées sur de petits programmes; on entend par là des projets à échelle réduite pour lesquels un seul programmeur fournit un effort intense durant une période limitée, ce que Conté et Basili définissent comme projet du micro-niveau [Conté 86].

Les premières expériences dans cette voie ont été réalisées par Halstead. Celui-ci a développé un modèle connu sous le nom de "software science" [Halstead 77]. Il avait constaté que compter le nombre de LOC, même en présence d'une définition précise de ce concept, était problématique. C'est pourquoi, il proposa de compter le nombre d'unités syntaxiques, c'est-à-dire le nombre d'opérateurs et opérandes.

Les opérateurs réalisent une action, par exemple les opérateurs standards ("+","-","*") mais aussi le ";" qui réalise une composition d'instructions et les mots réservés tels que le "if", "while", etc... Les opérandes désignent les variables et les constantes.

Les quatre grandes unités de base dans le modèle d'Halstead sont :

n_1 : le nombre d'opérateurs uniques (différents)

n_2 : le nombre d'opérandes uniques (différentes)

N_1 : le nombre total d'occurrences de chaque opérateur

N_2 : le nombre total d'occurrences de chaque opérande.

On exprime la taille du programme par :

$$N = N_1 + N_2$$

De cette manière, on réalise un raffinement du simple comptage du nombre de LOC, tout en gardant une forte corrélation avec l'effort de programmation nécessaire.

Cependant, il est intéressant de pouvoir estimer à priori la taille du logiciel. La valeur de N est une fonction de n_1 et n_2. Il n'est pas étonnant que la valeur de n_1 pour les langages de programmation de haut niveau soit relativement constante. Cependant, celle-ci dépend du langage choisi. Par conséquent, étant donné un langage, le nombre maximal d'opérateurs à utiliser est toujours fixé; ceux-ci sont énumérés dans la syntaxe du langage. De plus, un programme pas trop trivial emploiera au moins une fois chaque opérateur du langage. Une hypothèse supplémentaire peut être avancée : n_2 sera

principalement fixé par le nombre de variables (vars) présentes dans le programme. On en arrive alors à une relation de la forme :

la taille = 102 + 5.31 vars [Wang 84]

Chaque programme, dès lors, comprendra environ 100 LOC plus environ 5 LOC par variable présente. Les premières expériences démontrent qu'on peut obtenir des estimations précises de la taille et de l'effort d'un logiciel. Selon Wang, par des techniques top-down et des langages très typés comme Pascal, il est possible d'estimer la valeur de vars relativement tôt dans le développement du projet, de sorte qu'un tel modèle offre de bons résultats pour une estimation à priori.

La généralisation de ces résultats pour de gros projets n'est pas simple. En effet, dans ceux-ci la complexité des relations entre modules et la communication nécessaire entre les développeurs jouent un rôle non négligeable.

4. Les méthodes d'estimation pour les grands projets.

4.1.Les risques des méthodes intuitives réalisées par analogie et selon l'expérience.

Les méthodes d'estimation intuitives sont influencées politiquement car elles sont déterminées par des arguments autres que techniques. Ces arguments sont par exemple les raisonnements suivants :

- nous avons douze mois de temps libre pour ce projet, son développement durera donc douze mois. Ceci provient de la loi bien connue de Parkinson (cfr supra).

- nous savons que nos concurrents ont fait une offre d'un million; par conséquent, nous devons faire une offre de 900 000.

- nous estimons la durée du projet à une année, mais ce ne sera pas accepté par la direction; c'est pourquoi, nous présenterons délibérément une estimation de dix mois [Boehm 81].

A travers de tels raisonnements, Boehm a distingué plusieurs méthodes d'estimation intuitives :

- la méthode selon le principe de Parkinson,

- la méthode "price to win" (délibérément, on estime les coûts du logiciel les plus bas possibles de façon à décrocher le contrat),

- la méthode par analogie (cfr infra),

- le jugement expert (cfr infra).

A notre avis, les méthodes intuitives sont influencées par les personnes qui les réalisent, car elles ne sont pas des techniques purement objectives. En effet, les arguments politiques sont d'autant plus visibles si les estimations sont effectuées par des personnes directement impliquées dans la réalisation du projet, par exemple le chef de projet et les personnes qui en sont responsables. Ces estimations pourraient jouer rapidement un rôle d'outil de contrôle envers les personnes impliquées dans le développement.

4.1.1. Les bases de données.

Beaucoup de modèles décrits dans la section suivante sont basés sur les données de projets antérieurs. Il faut également remarquer que la collecte et l'usage de ces données peuvent être également guidés par des choix et des politiques. Pour ce manque d'objectivité, ces modèles pourraient paraître moins valables.

Pour contrecarrer ce problème de fiabilité des données, certaines organisations ont créé un service spécial attaché à l'estimation des projets et à la collecte des données caractéristiques des projets antérieurs. Les personnes de ce service sont indépendantes du service de développement, par conséquent elles sont moins influencées par des arguments politiques [van Vliet 87].

La création de ce service ne résout pas tous les problèmes car il reste à prouver que toute pression politique a disparu. De plus, van Vliet reconnaît qu'à côté de ce problème de fiabilité, il reste toujours des problèmes de clarté des données enregistrées, voire même un manque de données quantitatives sur les projets déjà réalisés ainsi qu'un manque de données sur les projets en cours de développement.

En général, les personnes ne voient pas l'intérêt de collecter des données sur un projet en plein développement, pourtant celles-ci permettraient de réajuster les premières estimations. van Vliet compare ceci au barbier du Moyen-Age qui était aussi médecin. Ce dernier faisait la réflexion suivante : "nous n'avons pas assez de temps pour prendre la température du patient car ses cheveux doivent être coupés". Pourquoi ne pas faire les deux en même temps, les deux n'étant pas mutuellement exclusifs ?

4.1.2. La méthode par analogie.

Etant donné le manque de fiabilité, de clarté et de complétude dans ces bases de données, les estimateurs les utilisent souvent de manière plus globale. Ils procèdent par analogie sans pour autant éplucher les moindres caractéristiques du projet. Par conséquent, ils se contentent souvent d'estimations basées sur des analogies globales. S'ils ont beaucoup d'expérience avec le type d'application, leurs estimations pourront être raisonnables. Par contre, l'effet d'apprentissage qui se produit pour tout nouveau type d'application peut mener à des estimations trop pessimistes. En effet, on peut s'attendre à ce que l'expérience pour un type d'application mène à une productivité plus grande. Ces propos de van Vliet sont illustrés dans le tableau 11.

Compiler	*Aantal mensmaanden*
1	72
2	36
3	14

TABLEAU 11 L'EFFET DE L'APPRENTISSAGE DANS L'ECRITURE D'UN COMPILATEUR

Il faut éviter de tomber dans le piège d'une estimation par analogie lorsque les particularités du projet et les circonstances dans lesquelles il sera développé ne sont pas analogues aux données de la BD. Oublier ceci aurait un effet significatif sur l'effort de développement malgré une certaine similitude. Par exemple, automatiser une bibliothèque de 25000 livres et automatiser une bibliothèque universitaire de plus d'un million de titres avec des exigences de prestation et un temps de développement plus court présentent des similitudes sans pour autant être analogues.

4.1.3. Le jugement expert.

4.1.3.1. La méthode Delphi.

On peut aussi impliquer plusieurs personnes dans une estimation pour arriver à un jugement expert. Chacun de ces experts réalise une estimation du coût sur base de son expérience. Cependant, un certain nombre de facteurs tels que les caractéristiques de personnalité et des propriétés du projet sont difficiles à pondérer. Par conséquent, l'estimation

qui en résulte ne sera jamais que ce que la qualité des experts ne permet.

Quand des estimations sont faites en groupe, on peut remarquer que certains membres ont une influence plus forte sur le résultat final. En effet, certaines personnes n'expriment pas leur avis ou sont très vite sous l'influence de "beaux parleurs". Le résultat final s'en trouvera défavorablement influencé. Pour éviter ce désagrément, on peut appliquer la méthode de Delphi en consultant plusieurs experts : chacun donne son avis par écrit et un "modérateur" rassemble les avis de chacun et les redistribue à tous les experts. Les noms ne sont pas mentionnés. Chacun, ensuite, sur base de cette information, fournit un nouveau jugement corrigé qui est à nouveau redistribué et ce processus continue jusqu'au consensus [Boehm 81].

4.1.3.2. Plusieurs estimations (la méthode de Putnam).

Une autre méthode pour obtenir des jugements plus fiables est de donner la possibilité aux experts de réaliser plusieurs estimations. Puisque tout le monde a tendance à considérer une estimation optimiste comme réaliste (on entend jamais dire qu'un programme s'est terminé plus tôt que prévu), Putnam a proposé une technique pour contrecarrer cette tendance [Putnam 79].

Cette technique, à l'origine destinée à l'estimation du nombre de LOC des composants d'un logiciel, semble utilisable dans un jugement expert. Etant donné une estimation optimiste (a), une estimation réaliste (m) et une estimation pessimiste (b), on obtient par l'utilisation d'une distribution béta, un effort attendu :

$$\text{l'effort} = (a + 4m + b) / 6.$$

Bien que cette estimation soit probablement meilleure que celle basée sur une moyenne de a et b, il faut se garder d'un trop grand optimisme à l'égard de sa justesse. En effet, les programmes ont tendance à être plus longs et à dépasser largement le temps de développement prévu.

4.2. Les premiers modèles algorithmiques.

Le contenu de la section précédente est clair : nous devons disposer de données sur une échelle historique étendue afin d'obtenir des estimations réellement fiables pour de nouveaux projets.

De plus, des prévisions sur le coût attendu peuvent être réalisées sur base des attributs mesurables d'un projet. De la même façon que l'on estime le coût de l'aménagement d'un jardin comme étant une combinaison pondérée d'un certain nombre d'attributs significatifs (grandeur du jardin, nombre de mètres carrés de pelouse, existence ou non d'un étang, ...), on

procédera à l'estimation du coût d'un projet informatique. C'est en réalité combiner les facteurs de coût.

Dans cette section, nous commencerons par citer les premières tentatives pour arriver à l'estimation du coût d'un logiciel par des modèles algorithmiques. Par modèle algorithmique, on entend une méthode réalisant l'estimation de coûts à l'aide de formules mathématiques obtenues par des études statistiques . De plus, ces méthodes proposent un certain nombre d'étapes à suivre, c'est-à-dire une démarche faisant penser à la structure d'un algorithme.

4.2.1. Le modèle de Nelson (modèle linéaire).

Dans [Nelson 66], un modèle linéaire est présenté sous la forme suivante :

$$\text{l'effort} = a_0 + \sum_{i=1}^{n} a_i \, x_i$$

où les a_i sont des constantes et les x_i des facteurs de coût qui ont un effet sur l'effort nécessaire.

Comme nous l'avons vu dans la partie traitant des facteurs de coût, ceux-ci ont une influence sur la productivité et donc sur l'effort. Par une analyse consciencieuse des données des projets antérieurs et des différentes combinaisons des facteurs, on peut arriver à un modèle d'estimation avec seulement un nombre limité de facteurs.

Ainsi Nelson arrive à un modèle constitué de 14 facteurs où E est l'estimation du nombre de M-H :

$$E = -33.63 + 9.15\,X_1 + 10.73\,X_2 + 0.51\,X_3 + 0.46\,X_4 + 0.40\,X_5 + 7.28\,X_6 - 21.45\,X_7 + 13.5\,X_8 + 12.35\,X_9 + 58.82\,X_{10} + 30.61\,X_{11} + 29.55\,X_{12} + 0.54\,X_{13} - 25.2\,X_{14}$$

Dans la figure 8, la signification de ces facteurs et leurs valeurs possibles sont mentionnées. Le domaine d'application de ce modèle est militaire. Il se peut très bien que certains facteurs n'aient aucune influence dans tout autre environnement.

Factor	*omschrijving*	*Mogelijke waarden*
x_1	Instabiliteit definitie van eisen	0-2
x_2	Instabiliteit ontwerp	0-3
x_3	Percentage wiskundige instructies	percentage
x_4	Percentage I/O-instructies	percentage
x_5	Aantal subprogramma's	aantal
x_6	Wel/geen hogere programmeertaal gebruikt	0-1
x_7	Wel/geen administratieve toepassing	1-0
x_8	Wel/geen stand-alone programma	0-1
x_9	Wel/niet eerste programma op deze machine	1-0
x_{10}	Apparatuur wel/niet tegelijk ontwikkeld	1-0
x_{11}	Wel/niet gebruik random-access device	1-0
x_{12}	Wel/geen verschillende ontwikkel/doelmachine	1-0
x_{13}	Aantal benodigde reizen	aantal
x_{14}	Wel/niet ontwikkeld door militaire organisatie	0-1

FIGURE 8 LES FACTEURS DU MODELE LINEAIRE DE NELSON

On peut remarquer que le facteur X_6 concernant le langage de programmation est invraisemblable car l'utilisation de l'assembleur plutôt qu'un langage de plus haut niveau se traduirait par environ 7 M-H supplémentaires sans tenir compte de la taille du projet (puisque la valeur ne varie que de 0 à 1). Egalement, l'utilisation d'une constante négative et de facteurs comptés négativement rendent l'estimation invraisemblable.

En général, les modèles linéaires ne fonctionnent pas bien. Il est certain qu'un grand nombre de facteurs influencent la productivité, mais il est très improbable qu'ils apparaissent indépendamment et linéairement [van Vliet 87].

Il est bon aussi de montrer le côté trop théorique de ce type de formule. Différentes constantes dans cette formule sont données avec une précision allant jusqu'à deux décimales. Une simple application de cette formule pourrait fournir un résultat de 97.32 M-H par exemple. La formule de Nelson est le résultat d'une analyse statistique des données de projets existants.

Selon van Vliet, il faut interpréter la formule de la façon suivante : pour une estimation A, la probabilité que ce projet coûte B M-H avec ((1-a)A <= B <= (1+a)A) est plus grande que p (avec par exemple, a = 0.2 et p = 0.9). Si la grandeur moyenne des hommes belges est de 1.8 mètres, ceci signifie aussi que la grandeur d'un belge quelconque se situe entre 1.6 et 2 mètres. La probabilité qu'il mesure exactement 1.8 mètres est

évidemment faible. Et il existe aussi une certaine probabilité qu'il soit plus petit que 1.6 mètres et plus grand que 2 mètres.

A l'aide de ce type de modèle, nous obtiendrons une estimation du coût comprise dans un intervalle et il reste une certaine probabilité que le coût sorte de cet intervalle.

4.2.2. Le modèle de Wolverton (modèle bottom-up).

La méthode bottom-up pourrait aider l'expert dans son estimation; on réalise d'abord une estimation de chaque module et le coût total est égal à la somme des coûts des modules avec une simple correction pour tenir compte de l'intégration du tout [Wolverton 74]. Wolverton décrit un modèle dans lequel une matrice des coûts est utilisée comme point de départ pour déterminer les coûts d'un module. Dans celle-ci, un nombre limité de types de modules est distingué, ainsi qu'un certain nombre de niveaux de complexité. Une telle matrice est proposée en figure 9.

moduultype	*complexiteit*				
	laag				hoog
	1	2	3	4	5
gegevensbeheer	22	26	31	37	44
geheugenbeheer	50	52	54	58	63
algoritme	12	17	29	54	102
gebruikersinterface	27	32	38	47	58
controle	40	50	60	70	80

FIGURE 9 MATRICE DE COUTS

Les éléments de la matrice sont ici les coûts en florins par LOC. Etant donné une matrice de coûts C et un module de type i, de complexité j et une taille estimée S, alors le coût estimé du modèle sera $M_k = S * C_{i,j}$

Ici aussi, il existe un certain nombre de griefs. L'utilisateur doit manipuler des définitions sujettes à interprétation pour déterminer les complexités des composants, ceci conduit à une belle dose d'incertitude. Certains facteurs ne sont pas pris en compte bien qu'on puisse leur attribuer une influence sur la productivité, par exemple, l'expérience de programmation et les caractéristiques du hardware sur lequel le système sera implémenté. Mais il faut ajouter que l'extension de cette matrice des coûts pour tenir compte de tels facteurs apporterait encore plus de subjectivité.

4.3. Les modèles les plus récents.

4.3.1. La formule générale.

Nous avions fait déjà remarquer que l'effort est fortement lié à la taille. Il existe différents modèles non linéaires où cette relation est exprimée sous la forme générale :

$$E = (a + b * KLOC^c)\ f(x_1, x_2, \ldots, x_n)$$

où KLOC est l'estimation de la taille exprimée en milliers de LOC,

E est l'effort en M-H,

a, b, c sont des constantes,

et $f(x_1, x_2, \ldots, x_n)$ est un facteur de correction qui dépend des grandeurs $x_1, \ldots, x_n$.

En général, par une analyse de régression à partir des données de projets antérieurs, on obtient la formule de base :

$$E = a + b\ KLOC^c.$$

Pour déterminer l'effort, les méthodes se basent sur la taille exprimée en LOC. L'estimation nominale des coûts est corrigée par un certain nombre de facteurs déterminant la productivité. Si l'un des facteurs utilisés est la qualité des programmeurs, par exemple, on peut apporter une correction de 1.5, 1.2, 1, 0.8 ou 0.6 pour respectivement une équipe très modérée, modérée, moyenne, bonne ou très bonne.

A la figure 10, sont reprises les formules de base les plus connues concernant la relation entre la taille estimée et l'effort nécessaire. Pour des raisons déjà citées, il n'est pas facile de les comparer, mais il très interessant de constater que la valeur de la constante c dans la plupart des modèles varie autour de 1. Nous pouvons l'expliquer par analogie avec un phénomène bien connu dans les sciences économiques.

Oorsprong	*Basisformule*
Halstead	$E = 0.7KLOC^{1.50}$
Walston-Felix	$E = 5.2KLOC^{0.91}$
Boehm	$E = 2.4KLOC^{1.05}$

FIGURE 10 QUELQUES FORMULES DE BASE POUR LA RELATION ENTRE LA TAILLE ET L'EFFORT

Dans une économie d'échelle, on part du principe qu'il est avantageux de produire des grandes quantités du même produit. Les coûts fixes sont alors répartis sur un plus grand nombre d'unités, ce qui entraîne une diminution du coût par unité. On obtient ainsi des rendements croissant. Par contre, on parle de rendement décroissant lorsqu'en produisant de grandes quantités, des coûts supplémentaires apparaissent au delà d'un certain seuil.

Dans le cas de la programmation, les LOC sont le produit. On peut alors dire que produire beaucoup de LOC entraîne moins de coûts par LOC; c'est pourquoi, on peut employer une formule comme celle de Walston et Félix (où c < 1). Cela peut être le cas parce que les coûts très onéreux des moyens d'aide (générateur de rapports ou de tests, les environnements de programmation) sont répartis sur plusieurs LOC. Par contre, des projets plus volumineux sont relativement plus coûteux : il y a des contretemps liés à la communication et à la gestion, les problèmes sont plus complexes, etc... Dans une telle situation, on arrive à des formules telles que celles de Boehm et Halstead (où c > 1) [van Vliet 87].

La plupart des modèles utilisent une valeur pour c supérieure à 1, ceci semble le plus plausible vu la complexité toujours plus croissante des programmes volumineux. Il est évident que la valeur de c pour les projets volumineux a une grande influence sur le calcul de l'effort.

Dans la figure 11, des valeurs de KLOC et de E pour quelques modèles cités sont présentées. On peut constater des divergences flagrantes : le modèle d'Halstead fournit pour de petits programmes le coût le plus bas; en ce qui concerne les projets de l'ordre du million de LOC, on constate que la formule d'Halstead donne un résultat dix fois plus grand que celui obtenu par la formule de Walston.

KLOC	$E = 0.7KLOC^{1.50}$	$E = 2.4KLOC^{1.05}$	$E = 5.2KLOC^{0.91}$
1	0.7	2.4	5.2
10	22.1	26.9	42.3
50	247.5	145.9	182.8
100	700.0	302.1	343.6
1000	22 135.9	3390.1	2792.6

FIGURE 11 L'EFFORT EN FONCTION DE LA TAILLE DANS QUELQUES MODELES DE BASE

Dans [Mohanty 81], 13 modèles quantitatifs ont été comparés pour un et un seul projet fictif. L'estimation du coût a varié de 362500 $ à 2776667 $. On ne peut pourtant pas conclure que ces modèles sont sans valeur. Il est très probable que de

grandes différences existent dans les types d'application sur lesquels sont basés ces modèles.

Il n'est pas conseillé de reprendre ces formules telles qu'elles sont présentées, car chaque environnement a sa spécificité et une orientation des paramètres du modèle est indispensable (calibration).

4.3.2. Le modèle de Walston et Félix [Walston 77].

L'équation de base de ce modèle respecte la formule générale :

$$E = 5.2\ KLOC^{0.91}$$

Des données relatives à un soixantaine de projets développés chez IBM sont à l'origine de son modèle. Ces projets différaient énormément par leur taille et par le langage de programmation utilisé. Il n'est donc pas étonnant que cette formule développée sur un ensemble partiel de ces projets, ne donne pas les meilleurs résultats [Conté 86]. Pour essayer d'expliquer les résultats divergents, Walston et Félix identifièrent 29 facteurs ayant une influence certaine sur la productivité. Pour chacun de ces facteurs, trois niveaux d'influence sont distingués : haut, moyen et bas.

Walston et Félix déterminèrent, par observation de 51 projets, la productivité attendue pour les trois niveaux de chaque facteur. Ces résultats sont mentionnés en figure 12. Par exemple, pour une interface utilisateur simple, la productivité s'élèvera à 500 LOC par M-H en moyenne; pour une complexité moyenne de l'interface, elle s'élèvera à 295 et pour une complexité forte à 124. Dans la dernière colonne, la valeur absolue de la différence entre la complexité simple et la complexité forte est donnée (PC : productivity change).

Selon Walston et Félix, un index de productivité (I) peut être obtenu pour un nouveau projet de la manière suivante :

$$I = \sum_{1}^{29} W_i\ X_i$$

Les poids W_i sont définis comme :

$$W_i = 0.5 \log (PC_i)$$

où PC_i est le changement de productivité pour le facteur i ($1 <= i <= 29$).

Pour le facteur 1 "interface utilisateur", cela vaut :

$$PC_i = 376 \text{ et } W_i = 1.29$$

Les variables X_i peuvent valoir 1, 0 ou -1 selon que le facteur correspondant a une complexité simple, moyenne ou forte (ce qui se traduit respectivement par une hausse de productivité, aucune influence sur la productivité ou une

baisse de productivité). L'index de productivité ainsi obtenu peut être réduit en une productivité attendue.

Le nombre de facteurs déterminant la productivité dans ce modèle est très grand et il n'est pas évident de discerner les interrelations entre ces facteurs. Les trois niveaux de complexité semblent offrir trop peu de possibilités pour faire face à tous les cas de la pratique.

Cependant, l'approche de Walston et Félix, et la liste des 29 facteurs influençant la productivité ont joué un rôle important pour les recherches qui ont suivi.

Question or Variable	*Response Group / Mean Productivity (DSI/MM)*			*Productivity Change (DSI/MM)*
Customer interface complexity	Normal 500	Normal 295	> Normal 124	376
User participation in the definition of requirements	None 491	Some 267	Much 205	286
Customer originated program design changes	Few 297		Many 196	101
Customer experience with the application area of the project	None 318	Some 340	Much 206	112
Overall personnel experience and qualifications	Low 132	Average 257	High 410	278
Percentage of programmers doing development who participated in design of functional specifications	< 25% 153	25 - 50% 242	> 50% 391	238
Previous experience with operational computer	Minimal 146	Average 270	Extensive 312	166
Previous experience with programming languages	Minimal 122	Average 225	Extensive 385	263
Previous experience with application of similar or greater size and complexity	Minimal 146	Average 221	Extensive 410	264
Ratio of average staff size to duration (people/month)	< 0.5 305	0.5 - 0.9 310	> 0.9 173	132
Hardware under concurrent development	No 297		Yes 177	120
Development computer access, open under special request	0% 226	1 - 25% 274	> 25% 357	131
Development computer access, closed	0 - 10% 303	11 - 85% 251	> 85% 170	133
Classified security environment for computer and 25% of programs and data	No 289		Yes 156	133

Question or Variable	*Response Group / Mean Productivity (DSL/MM)*			*Productivity Change (DSL/MM)*
Structured programming	0-33% 169	34-66% –	66% 301	132
Design and code inspections	0-33% 220	34-66% 300	> 66% 339	119
Top down development	0-33% 196	34-66% 237	> 66% 321	125
Chief programmer team usage	0-33% 219	34-66% –	> 66% 408	189
Overall complexity of code developed	< Average 314		> Average 185	129
Complexity of application processing	< Average 349	Average 345	> Average 168	181
Complexity of program flow	< Average 289	Average 299	> Average 209	80
Overall constraints on program design	Minimal 293	Average 286	Severe 166	107
Program design constraints on main storage	Minimal 391	Average 277	Severe 193	198
Program design constraints on timing	Minimal 303	Average 317	Severe 171	132
Code for real-time or interactive operation, or executing under severe timing constraint	< 10% 279	10-40% 337	> 40% 203	76
Percentage of code for delivery	0-90% 159	91-99% 327	100% 265	106
Code classified as non-mathematical application and I/O formatting programs	0-33% 188	34-66% 311	67-100% 267	79
Number of classes of items in the data base per 1000 lines of code	0-15 334	16-80 243	> 80 193	141
Number of pages of delivered documentation per 1000 lines of delivered code	0-32 320	33-88 252	> 88 195	125

FIGURE 12 LES 29 FACTEURS DE WALSTON ET FELIX

4.3.3. Le modèle Cocomo de Boehm [Boehm 81].

Cocomo (constructive cost model) est l'un des modèles algorithmiques le plus traité en profondeur et le plus documenté. Dans sa forme la plus simple, la formule de l'effort pour une taille donnée respecte la formule générale où les constantes dépendent du type d'applications. Boehm en distingue trois :

- l'application organique où les constantes b et c de la formule générale valent 2.4 et 1.05,
- l'application semi-détachée où b = 3 et c= 1.12,
- l'application imbriquée où b = 3.6 et c = 1.2.

Par application organique, il faut entendre une application dans laquelle une petite équipe travaille dans un environnement habituel. Cette équipe a beaucoup d'expérience pour ce type d'application. Il existe une collaboration rapide et efficace entre les développeurs et les projets sont de petites tailles.

Par application imbriquée, il faut entendre des systèmes plus complexes pour lesquels l'environnement pose de très fortes exigences. Le produit sera intégré dans un environnement peu flexible.

Les applications semi-détachées se situent entre les applications organiques et imbriquées. Le projet a une ampleur importante mais pas excessive.

L'estimation de l'effort pour ces trois types d'applications est donnée en figure 13, en fonction de différentes tailles variant de 1 à 1 000 LOC.

KLOC	*Inspanning in mensmaanden*		
	organisch ($E = 2.4KLOC^{1.05}$)	*mengeling* ($E = 3.0KLOC^{1.12}$)	*ingebed* ($E = 3.6KLOC^{1.20}$)
1	2.4	3.0	3.6
10	26.9	39.6	57.1
50	145.9	239.4	392.9
100	302.1	521.3	904.2
1000	3390	6872	14 333

FIGURE 13 L'EFFORT EN FONCTION DE LA TAILLE DANS LE MODELE COCOMO DE BASE

Ce modèle de base fournit une estimation brute du coût. Boehm propose deux modèles plus compliqués. Il s'agit des modèles cocomo intermédiaire et détaillé. Dans ceux-ci,

l'estimation brute est corrigée par 15 facteurs, pour lesquels cinq niveaux de complexité sont définis (cfr figure 14).

Factoren	*Score*				
	zeer laag	*laag*	*geen*	*hoog*	*zeer hoog*
Produkt-factoren:					
vereiste betrouwbaarheid van de programmatuur	.75	.88	1.00	1.15	1.40
grootte van de databank		.94	1.00	1.08	1.16
complexiteit van het produkt	.70	.85	1.00	1.15	1.30
Computer-factoren:					
beperkingen voor wat betreft executietijd			1.00	1.11	1.30
geheugenbeperkingen			1.00	1.06	1.21
frequentie waarmee veranderingen in hardware optreden		.87	1.00	1.15	1.30
snelheid waarmee jobs verwerkt worden		.87	1.00	1.07	1.15
Personeels-factoren:					
capaciteit van de analisten	1.46	1.19	1.00	.86	.71
ervaring met de toepassing	1.29	1.13	1.00	.91	.82
capaciteit van de programmeurs	1.42	1.17	1.00	.86	.70
ervaring met de onderliggende hardware/software	1.21	1.10	1.00	.90	
ervaring met de te gebruiken programmeertalen	1.14	1.07	1.00	.95	
Project-factoren:					
gebruik van geavanceerde programmeertechnieken	1.24	1.10	1.00	.91	.82
gebruik van hulpmiddelen	1.24	1.10	1.00	.91	.83
eisen aan projectduur gesteld	1.23	1.08	1.00	1.04	1.10

FIGURE 14 LES FACTEURS DE COUT DANS COCOMO

Si l'effort nécessaire de développement est de 40 M-H (estimation brute) et si la complexité de programmation est basse, alors le modèle propose un facteur de correction égal à 0.8, ce qui entraîne une diminution de l'effort de six M-H.

Le modèle détaillé apporte un raffinement supplémentaire en répartissant l'influence d'un facteur sur chacune des phases du développement. Pour chaque facteur, Boehm donne une table dans laquelle le taux d'influence varie en fonction de la phase de développement. Par exemple, un haut temps de réponse aura relativement peu d'effet sur les phases de définition et d'analyse du problème; par contre il nécessitera plus de temps durant les phases de programmation et de tests. En fait, de cette manière, il est possible d'estimer l'effort nécessaire pour chacune des phases.

De plus, Boehm réalise dans le modèle détaillé des estimations au niveau du module ou du sous-système. Malgré ce degré de détails, selon Kitchenham dans [Kitchenham 85], il existe assez bien de différences entre l'effort réel et l'effort estimé par cocomo. Il faut cependant insister sur le fait que tous les détails de l'étude de Boehm ont été publiés et sont disponibles dans [Boehm 81].

4.3.4. Le modèle de Putnam/Norden [Putnam 79].

Norden découvrit déjà dans les années 60 un comportement très régulier de la force de travail dans le temps de développement. La répartition caractéristique de la force de travail est bien approchée par la distribution de Rayleigh (cfr

figure 15). Sur cette base, Putnam a développé un modèle dans lequel la force de travail (ft) est donnée au moment t par l'équation suivante :

$$ft(t) = 2\ K\ a\ t\ e^{-a t^2}$$

où a est un facteur d'accroissement qui détermine la pente de la courbe,

K est le nombre total de M-H (y compris la phase de maintenance), il est égal à la surface de la région située sous la courbe.

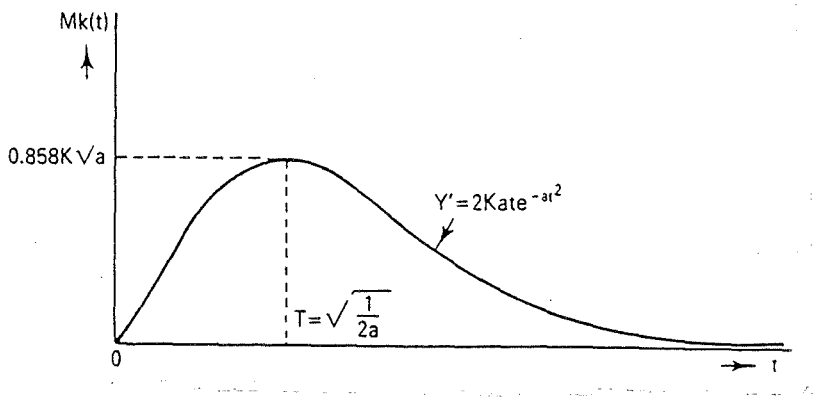


FIGURE 15 LA DISTRIBUTION DE RAYLEIGH

La forme de cette courbe peut être expliquée selon la théorie de Parr [Parr 80] : soient un projet constitué d'un ensemble de données sur un problème à résoudre, W(t) la part du problème résolue au temps t et p(t) la capacité disponible au temps t pour la résolution du problème, alors l'avancement au temps t est proportionnel au produit de la capacité présente et de la fraction de problèmes non résolus. Si la quantité de travail est égal à 1, alors :

$$(d\ W\ /\ d\ t) = p(t)\ (1-W(t))$$

Après intégration, on obtient :

$$W(t) = 1-\exp\left(-\int_{}^{t} p(\alpha)\ d(\alpha)\right)$$

Si on accepte que la capacité est bien approchée par la relation p(t) = at (ceci veut dire que la capacité pour résoudre le problème croît linéairement dans le temps), l'avancement est donné par la distribution de Rayleigh :

$$(d\ W\ /\ d\ t) = a\ t\ e^{-a t^2/2}$$

L'intégration de la première relation donnée pour ft fournit une relation pour l'effort cumulé I :

$$I(t) = K\ (1 - e^{-at^2})$$

En particulier, on a $I(\infty) = K$

Si T est le moment où la courbe de Rayleigh atteint son maximum, alors a vaut à ce moment : $1/(2T^2)$

Ce temps T représente le moment où le programme sera livré. La surface sous la courbe entre le temps 0 et le temps T est alors une bonne approximation de l'effort nécessaire jusqu'à la livraison du programme.

Alors, E = I(T) = 0.3945 K

Ceci se rapproche de la règle très connue : 40 % de développement pour 60 % de maintenance.

Ces équations citées sont, avec beaucoup d'autres, reprises dans un produit commercial appelé SLIM [Putnam 80]. Différentes études [Conté 86] et [DeMarco 82] montrent que le modèle de Putnam fonctionne bien pour de gros projets. Dans [Parr 80], une variante de ce modèle est décrite. Elle se base sur l'observation suivante : à la réception d'un projet, les personnes impliquées dans le développement ont souvent une certaine expérience du type d'applications à développer. Dans ce cas, la courbe ne débute pas à l'origine des axes (cfr figure 16).

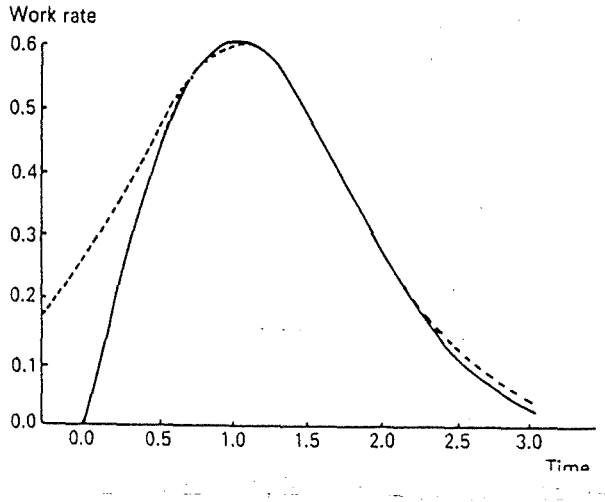


FIGURE 16 LA COURBE DE PARR

4.3.5. Le modèle de DeMarco [DeMarco 82]

Les modèles comme Cocomo fournissent une estimation de l'effort en fonction du nombre de LOC. Ce nombre de LOC n'est cependant pas facile à estimer dans les premières phases du développement. Dans [DeMarco 82], un modèle est proposé dans lequel une estimation de l'effort peut être réalisée dès la définition de projet.

Pour la modélisation d'une application, il emploie le diagramme de flux de données; de cette manière, il exprime les transformations des différentes structures de données qui, une fois rassemblées, fournissent un véritable réseau de fonctions à exécuter.

Le modèle de DeMarco est basé sur le nombre de primitives fonctionnelles (pf) dans l'interface homme-machine de l'application. Ces pf sont les fonctions du diagramme de flux des données. Naturellement, certaines primitives sont plus complexes que d'autres, de sorte que l'on doit apporter certaines corrections pour arriver à une mesure plus uniforme. A chacune de ces fonctions ou encore transformations, des éléments d'inputs et d'outputs sont concernés. DeMarco approche maintenant le volume relatif (contenu informationnel) d'une transformation par une fonction du nombre d'éléments concernés par cette transformation (tc). Ceci donne :

$$volume(pf) = K\ tc\ \log_2 (tc)$$

où K est une constante

On tient compte de la complexité des pf par un facteur multiplicatif de correction. Chaque pf est portée dans une des catégories suivantes :

- fonctions qui divisent ou combinent des données d'entrée (le facteur multiplicateur est 0.6),
- fonctions qui mettent à jour des données (0.3),
- fonctions qui analysent des données et qui sur cette base décident d'une action (1),
- fonctions qui évaluent les entrées de l'interface homme-machine (0.8),
- fonctions qui contrôlent la cohérence interne (1),
- fonctions pour la manipulation de textes (1),
- fonctions qui synchronisent l'interaction avec les utilisateurs (1.5),
- fonctions qui génèrent des sorties (1),
- fonctions qui exécutent de simples calculs (0.7),
- fonctions qui exécutent des calculs compliqués (2).

Au lieu du nombre brut de pf, DeMarco utilise une approche corrigée (pf') en fonction du volume et de la complexité :

$$pf' = \sum_i \alpha_i * tc_i * \log_2 (tc_i)$$

où α_i est le facteur multiplicatif de la catégorie à laquelle la fonction i appartient.

Finalement, il reste encore une relation entre pf' et l'effort nécessaire E :

$$E = a\ (pf')^b$$

où a et b sont des constantes obtenues par analyse des projets antérieurs.

On peut remarquer que DeMarco est le seul auteur de méthode d'estimation à ne pas proposer une valeur pour les constantes a et b car elles doivent être déterminées dans chaque environnement.

Ce modèle nous a semblé très intéressant parce qu'il propose une alternative aux LOC, cependant nous n'avons pas trouvé plus d'informations à son sujet.

4.3.6. La méthode des points de fonction [Albrecht 79].

La méthode des points de fonction (pf) est une méthode qui, comme celle de DeMarco, évite les problèmes de détermination du nombre de LOC. Cette méthode s'applique en particulier aux applications administratives où l'architecture des données joue un rôle important. La méthode est moins destinée aux projets dont l'architecture des données joue un rôle de subordination et où l'accent est plutôt porté sur des traitements complexes.

Dans ce modèle, les cinq unités suivantes jouent un rôle important :

- les "inputs" (inp): les informations en provenance de l'utilisateur, nécessaires à l'exécution des traitements,
- les "outputs" (out) : les résultats des traitements destinés à l'utilisateur,
- les "inquiries"(inq) : les combinaisons d'inputs/outputs ne réalisant qu'une simple consultation des données de l'application,
- les "master files" (mf):les données sur lesquelles s'exécutent les traitements,
- les "interfaces" (int) : les données de l'application à mesurer qui sont transférées à d'autres applications ou les données d'autres applications transférées à l'application à mesurer.

Par essais et erreurs, des poids ont été attribués à ces cinq unités en fonction de leur complexité. Le nombre de points de fonction est alors une moyenne pondérée des cinq unités :

le nombre de pf = 4 inp + 5 out + 4 inq + 10 mf
+ 7 int

Ce nombre de pf est en réalité le nombre de pf brut qui sera corrigé par 14 facteurs de coût déterminant la complexité des traitements.

Albrecht a converti un pf en un certain nombre de LOC en fonction du langage : un pf représente en moyenne 65 LOC en PL/1 et 100 LOC en COBOL.

Il existe un manque de précision dans les définitions des cinq unités; il est parfois bien difficile de distinguer, par exemple un masterfile d'une interface. Il serait donc nécessaire de définir plus précisément les concepts de cette méthode.

5. La distribution de la force de travail dans le temps.

Après avoir obtenu une estimation du nombre total de M-H nécessaires, il faut encore le répartir dans le temps. Pour un projet estimé à 20 M-H, on a en ce qui concerne la répartition des personnes dans le temps, les possibilités suivantes :

- vingt personnes travaillant un mois au projet
- quatre personnes travaillant cinq mois au projet
- une personne travaillant un mois au projet.

Chacune de ces possibilités n'est pas nécessairement bonne; aucune n'est adaptée car chaque moment ne requiert pas la même quantité de personnes. Par la courbe de Rayleigh, nous avons vu un besoin croissant de personnes durant la phase de développement d'un projet. Ce temps de développement (t) est en général estimé par les différentes méthodes de manière uniforme, comme le montre la figure 17 [van Vliet 87].

Walston-Felix	$T = 2.5E^{0.35}$
COCOMO (organisch)	$T = 2.5E^{0.38}$
Putnam	$T = 2.4E^{1/3}$

FIGURE 17 LE TEMPS DE DEVELOPPEMENT EN FONCTION DE L'EFFORT

Les valeurs obtenues pour t représentent un temps de développement nominal; il est donc intéressant de considérer que le temps de développement peut être diminué de manière raisonnable par une augmentation du nombre de personnes. Cela revient à augmenter, selon Putnam, la constante a, c'est-à-dire le facteur d'accroissement qui représente la pente de la courbe de Rayleigh. Le sommet de celle-ci glisse alors vers la gauche et monte simultanément : nous avons ainsi une augmentation plus rapide de la force de travail nécessaire en début de développement. Cependant, un tel glissement n'est pas certain : différentes études ont démontré que la productivité individuelle diminue lorsque le nombre de personnes augmente (cfr section "facteurs de coût").

6. Conclusion.

Le modèle le plus adapté pour réaliser l'estimation des coûts d'un projet n'est pas encore inventé. En effet, les modèles discutés fournissent des résultats fort distincts, ce qui a été une fois de plus prouvé par l'étude de Rubin [Rubin 85]. Des enquêtes réalisées actuellement, on peut retirer qu'un modèle ne sera jamais suffisamment général pour être appliqué à tout environnement. Le nombre de paramètres influençant la productivité est trop important. Cependant, une organisation peut élaborer elle-même un modèle qui sera bien mieux adapté pour l'estimation de ses projets. Il lui faut pour cela construire une base de données qui lui est propre. Pour ce faire, elle pourrait commencer par utiliser, par exemple, le modèle cocomo pour attribuer à chaque facteur de coût une influence propre à l'organisation. Dans [Bailey 81], une telle procédure est décrite.

Il reste malgré tout encore quelques problèmes :

- bien qu'un modèle comme cocomo semble objectif parce qu'il utilise un grand nombre de facteurs, il introduit une grande part de subjectivité. Dans [Jones 86], sur base d'une analyse historique, celui- ci a dénombré vingt facteurs influant avec certitude sur la productivité et vingt-cinq autres qui le font de manière probable. Selon lui, il serait encore possible de réduire ce nombre de facteurs influants mais tout ceci reste subjectif.

- les modèles algorithmiques sont basés sur les données de projets antérieurs qui, souvent, ne comprennent pas l'influence de techniques récentes telles que le prototypage, les programmations orientées objet, etc...

- les facteurs influençant le développemnt du projet sont repris dans tous les modèles tandis que les facteurs concernant l'aspect maintenance sont rarement pris en considération, par exemple, la documentation requise, les voyages nécessaires, etc... Pourtant, ceux-ci ont une influence significative sur les coûts.

- étant donné un budget de 250000 francs pour l'automatisation d'une bibliothèque, quelles sont les possibilités existantes ? Quelle qualité peut-on espérer en fonction de ce budget, par exemple, pour les interfaces, la rapidité des transactions, la fiabilité, etc...? Pour pouvoir répondre à ce type de questions, il faut pouvoir réaliser des analyses de sensibilité des estimations. Mais étant donné le manque de clarté concernant la pertinence des facteurs et leur influence mutuelle, de telles analyses sont compromises.

- une estimation du coût d'un logiciel à priori peut avoir une influence sur le développement [Abdel-Hamid 86]. Selon la loi de Parkinson, nous savons déjà que le temps disponible estimé pour le développement d'un logiciel est complètement utilisé, même s'il était possible de réaliser plus vite ce dernier. Par conséquent, des estimations trop pessimistes ont tendance à allonger inutilement le temps de développement.

CHAPITRE 5.

PRESENTATION DES METHODES D'ESTIMATION LES PLUS COURANTES.

1. Introduction.

Etant donné le caractère générale des chapitres 2 et 4, délibérément choisi pour tracer l'évolution des méthodes d'estimation, il nous a semblé utile, pour les personnes intéressées de décrire de manière plus détaillée les méthodes les plus courantes, à savoir celles de Walston et Félix, d'Halstead, de Putnam et de Boehm. Pour chacune d'entre elles, nous avons tenté d'expliquer les motivations de leur auteur.

Cependant, la lecture de ce chapitre n'est pas utile pour la bonne compréhension du travail.

NB : la méthode des points de fonction est largement décrite dans la deuxième partie du travail.

2. La méthode de Walston et Félix. [Walston et Félix 77]

2.1. Motivations.

Les recherches de Walston et Félix se sont orientées vers une méthode d'estimation de la productivité de programmation. Elle a pour but, la mesure du taux de production de LOC par projet. Cette mesure est influencée par un certain nombre de conditions et de besoins propres au projet.

Ces recherches commencèrent en 1972 quand ils décidèrent d'évaluer les effets d'une programmation structurée sur le processus de développement d'un logiciel. Pour ce faire, un programme rigoureux fut établi pour mesurer les méthodologies logicielles existantes, et dés lors il fut possible d'évaluer les changements de productivité résultant de l'introduction d'une nouvelle méthodologie.

La phase initiale de ce programme fut l'identification des variables à mesurer afin de créer un questionnaire et de développer un système pour analyser les données collectées par ce questionnaire. C'est ainsi qu'ils élaborèrent leur base de données.

Par la suite, ils comprirent que leurs objectifs pourraient changer au fur et à mesure que les recherches s'effectuaient et que les données s'accumulaient. Leurs objectifs devinrent :

- fournir des données pour évaluer les nouvelles technologies de programmation,

- fournir un support pour faire des offres et établir des contrats de développement,

- rassembler et conserver des enregistrements historiques de travaux terminés,

- fournir des informations au management,

- développer une terminologie commune pour préciser tous les concepts employés dans le jargon informatique.

2.2. Explication du modèle.

Les données contenues dans les questionnaires remplis à des périodes prescrites du développement sont stockées dans la base de données où elles sont accessibles pour répondre à différentes questions, pour préparer des rapports et pour procéder à des études analytiques. Les périodes prescrites sont :

- en fin de définition de projet,

- en fin d'analyse conceptuelle,

- en fin d'analyse fonctionnelle,

- en fin de tests,

- tous les trois mois après le passage en production et durant la maintenance.

La base de données est structurée de telle manière qu'elle puisse contenir les informations décrite au tableau 12.

Report Name	*Nature of the Report*
Programming Project Summary	Detailed report on the software development environment, the product (including errors), resources, and schedule.
Software Development Report	Detailed report on the software development environment, the product (including changes and errors), resources and schedule.
Monthly Software Development Report	One report on product, resource, and schedule status. Changes in software development environment are noted.
Software Service Report	Report on project size and on the software service environment.
Quarterly Software Service Report	Detailed data on product being serviced, resource status, and changes in software service environment.

TABLEAU 12 LES RAPPORTS DE MESURES DE LOGICIELS

Soixante projets furent stockés dans la base de données. Ils représentent un large éventail de technologies de programmation; comme le montre le tableau 13. Le nombre de LOC varie de 4000 à 467000, l'effort de 12 à 11758 M-H. Ces programmes contiennent des traitements en temps réel, interactifs, des générateurs de rapports et des contrôles de la base de données. Certains programmes sont soumis à des temps de réponse sévères et/ou à des limitations d'espace mémoire. 28 langages différents et 66 types d'ordinateurs sont représentés par ces projets.

A. Small less-complex systems
- Batch storage and retrieval
- Batch inventory
- Batch information management
- Batch languages preprocessor and information management
- Batch reporting
- Batch financial information
- Batch scientific processing simulation
- Batch utility
- Batch operating system exerciser

B. Medium less-complex systems
- Special-purpose data management (2)
- Batch storage and retrieval (2)
- Process control simulation
- Batch reporting
- Batch data-base utility (2)
- On-line scientific processing simulation
- Batch on-line scientific information management
- On-line business information management
- On-line storage and retrieval
- Batch hardware test support
- Batch scientific algorithm feasibility (3)
- Interactive scientific processing (2)
- System test support
- Batch planning (3)
- Batch military information management
- Special-purpose operating system

C. Medium complex systems
- Real-time, special-purpose system exerciser
- Special-purpose operating system (2)
- Batch information modification
- Batch information conversion
- Data management
- Sensor-based mission control
- On-line scheduling
- Sensor-based mission simulation
- Interactive scientific processing
- Process control (3)
- On-line graphics
- System performance monitoring and measurement (3)
- Terminal data management
- Interactive information conversion
- Operating system extensions

D. Large complex systems
- Sensor-based mission monitoring and control
- Interactive information acquisition
- Process control
- Sensor-based system exerciser (3)
- Sensor-based mission processing and communication (2)

TABLEAU 13 CARACTERISATION DES PROGRAMMES DANS LA BASE DE DONNEES DES PROJETS

Ce système permet de fournir de nombreux services : des réponses à différentes questions, des analyses de données, des estimations de productivité pour des nouveaux projets ou des projets en cours. Il existe deux types de questions: soit interroger la base de données pour obtenir des informations sur un projet bien déterminé, soit effectuer des requêtes d'un type plus général. Par exemple :

- le nombre de LOC,

- la part de LOC reprise d'un autre logiciel,

- le nombre de pages de documentation,

- le type de langage utilisé,

- la durée du projet,
- l'effort total de développement,
- l'utilisation de technologies de programmation :
- la programmation structurée,
- le développement top-down,
- l'effet d'une nouvelle technologie de programmation sur la productivité,
- le calcul de productivité,
- le nombre moyen de personnes pour réaliser le projet.

En ce qui concerne les analyses, on parlera plutôt de calculs statistiques (déviation, écart-type, moyenne,...) que de caractéristiques de distribution des variables.

Pour chacun des 60 projets, Walston et Félix ont retenu la taille du logiciel et l'effort total correspondant. Ils ont placé ces deux données sur un système d'axes orthogonaux où chacun des points symbolise un projet (cfr figure 18).

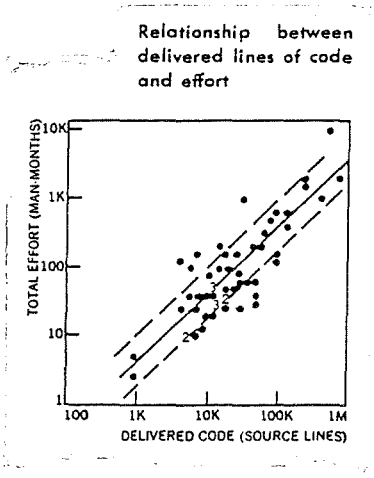


FIGURE 18 RELATION ENTRE LIGNES DE CODE ET EFFORT

Ces données ont été analysées de façon statistique dans le domaine log-log afin de les rendre approximativement linéaires. Si on pose que x = log X et y = log Y où x et y sont l'effort

et la taille du logiciel, et sachant que l'équation d'une droite est de la forme x = ay + b, on obtient par remplacement de x et y, l'équation suivante :

$$\log X = a \log Y + b$$

$$= \log Y^a + \log e^b$$

$$= \log(Y^a * e^b)$$

$$e^{\log X} = e^{\log(Y^a * e^b)}$$

$$X = Y^a * e^b$$

$$X = B\ Y^a \text{ où } B = e^b$$

Ce développement amène à la formule générale de l'effort en fonction de la taille :

$$E = b\ KLOC^c$$

La technique des moindres carrés appliquée à ces données a permis de dégager la formule suivante :

$$\text{l'effort} = 5.2\ KLOC^{0.91}$$

On constate une relation presque linéaire entre la taille et l'effort de développement. Cependant, étant donné un exposant inférieur à 1, l'effort augmente moins que proportionnellement à une augmentation de la taille.

Les lignes en pointillés indiquent l'erreur standard des estimations.

Pour identifier les causes de ces écarts, Walston et Félix ont sélectionné 68 variables. Et après analyse, 29 de ces variables montrèrent une corrélation significative avec la productivité et par conséquent, avec l'effort. Ces variables ont été retenues pour être utilisées dans les estimations (cfr le tableau 14). Suivant le degré de complexité (inférieur à la normal, normal, supérieur à la normal), Walston et Félix leur a attribué une productivité moyenne. Pour chaque variable un "productivity change", c'est-à-dire la différence en valeur absolue entre le niveau de complexité le plus bas et le niveau le plus élevé.

TABLEAU 14 LES 29 VARIABLES DE WALSTON ET FELIX

Question or Variable	*Response Group / Mean Productivity (DSL/MM)*			*Productivity Change (DSL/MM)*
Customer interface complexity	Normal 500	Normal 295	> Normal 124	376
User participation in the definition of requirements	None 491	Some 267	Much 205	286
Customer originated program design changes	Few 297		Many 196	101
Customer experience with the application area of the project	None 318	Some 340	Much 206	112
Overall personnel experience and qualifications	Low 132	Average 257	High 410	278
Percentage of programmers doing development who participated in design of functional specifications	< 25% 153	25–50% 242	> 50% 391	238
Previous experience with operational computer	Minimal 146	Average 270	Extensive 312	166
Previous experience with programming languages	Minimal 122	Average 225	Extensive 385	263
Previous experience with application of similar or greater size and complexity	Minimal 146	Average 221	Extensive 410	264
Ratio of average staff size to duration (people/month)	< 0.5 305	0.5–0.9 310	> 0.9 173	132
Hardware under concurrent development	No 297		Yes 177	120
Development computer access, open under special request	0% 226	1–25% 274	> 25% 357	131
Development computer access, closed	0–10% 303	11–85% 251	> 85% 170	133
Classified security environment for computer and 25% of programs and data	No 289		Yes 156	133

Question or Variable	*Response Group / Mean Productivity (DSL/MM)*			*Productivity Change (DSL/MM)*
Structured programming	0–33% 169	34–66% –	66% 301	132
Design and code inspections	0–33% 220	34–66% 300	> 66% 339	119
Top down development	0–33% 196	34–66% 237	> 66% 321	125
Chief programmer team usage	0–33% 219	34–66% –	> 66% 408	189
Overall complexity of code developed	< Average 314		> Average 185	129
Complexity of application processing	< Average 349	Average 345	> Average 168	181
Complexity of program flow	< Average 289	Average 299	> Average 209	80
Overall constraints on program design	Minimal 293	Average 286	Severe 166	107
Program design constraints on main storage	Minimal 391	Average 277	Severe 193	198
Program design constraints on timing	Minimal 303	Average 317	Severe 171	132
Code for real-time or interactive operation, or executing under severe timing constraint	< 10% 279	10–40% 337	> 40% 203	76
Percentage of code for delivery	0–90% 159	91–99% 327	100% 265	106
Code classified as non-mathematical application and I/O formatting programs	0–33% 188	34–66% 311	67–100% 267	79
Number of classes of items in the data base per 1000 lines of code	0–15 334	16–80 243	> 80 193	141
Number of pages of delivered documentation per 1000 lines of delivered code	0–32 320	33–88 252	> 88 195	125

Les 29 variables sont alors combinées en un index basé sur l'effet de chacune des variables sur la productivité. Cet index est calculé de la manière suivante :

$$I = \sum_{1}^{29} W_i \, X_i$$

où I est l'index de productivité

W_i = 0.5 log PC_i ou le poids de la variable sur la productivité

PC_i = le changement de productivité de la variable i

X_i = 1, 0 ou -1 suivant que le degré de complexité est respectivement inférieur à la normale, normal ou supérieur à la normale. C'est donc le sens dans lequel le poids de la variable influera sur la productivité. Ainsi, si pour une variable i, X_i est égal à +1, ceci signifie que le degré de complexité de la variable est inférieur à la normale, le projet en est plus facile, donc la productivité est plus élevée; ce qui se traduit par un index de productivité qui augmente. Si X_i est égale à 0, cela signifie qu'elle n'a pas d'influence sur la productivité. Si X_i est égale à -1, cela signifie une correction de la productivité à la baisse suite à la complexité élevée de la variable.

Pour 51 projets contenus dans la BD, ils ont calculé l'index de productivité. Disposant également de la productivité pour chacun des projets, ils ont pu établir la figure 19 exprimant la relation entre la productivité et l'index de productivité.

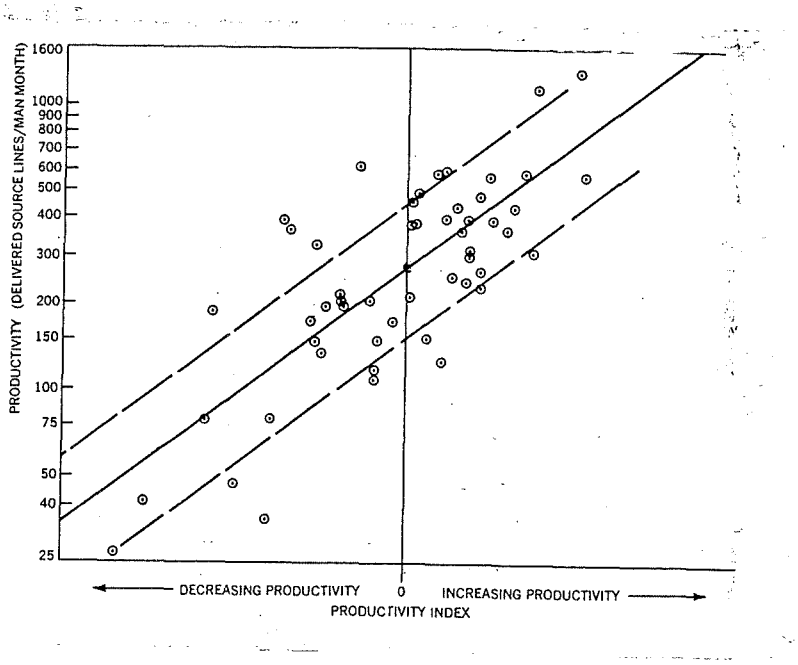


FIGURE 19 RELATION ENTRE LA PRODUCTIVITE ET L'INDEX DE PRODUCTIVITE POUR 20 VARIABLES

En consultant le graphique et pour un index de productivité égal à 0, on peut constater une productivité moyenne pour les 51 projets d'environ 280 LOC par M-H. On peut dire également que l'index de productivité (supérieur ou inférieur à 0) constitue un facteur de correction par rapport à la moyenne. De plus, sur cette base, pour estimer la productivité d'un nouveau projet, il suffit de calculer son index de productivité ou encore de déterminer l'influence propre des 29 variables pour en connaître la productivité.

Exemple (cfr figure 20):

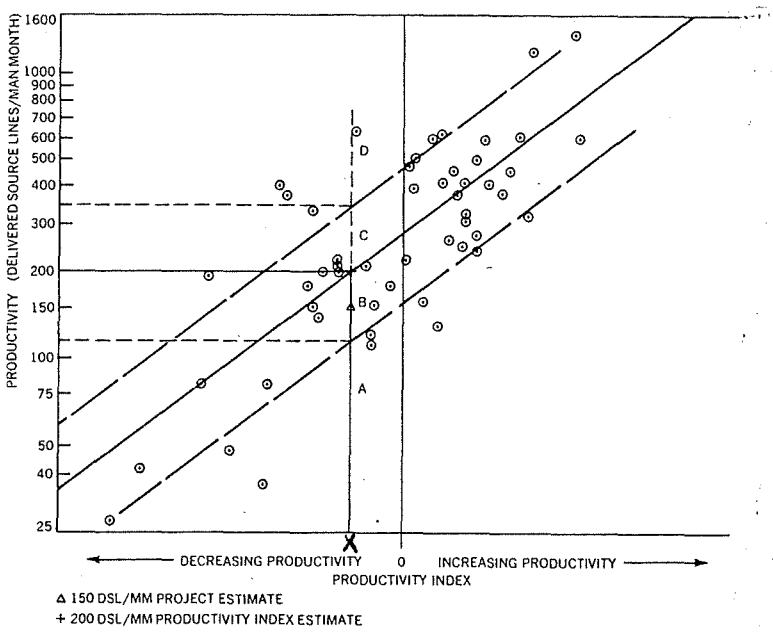


FIGURE 20 LA PRODUCTIVITE ESTIMEE POUR UN PROJET HYPOTHETIQUE

Pour un nouveau projet ayant un index de productivité égal à x, la productivité attendue sera de 200 LOC par M-H avec un intervalle d'erreur standard de 115 à 340 LOC par M-H. En connaissant la productivité, on pourra en déduire la taille et l'effort de développement.

En utilisant la même technique statistique que celle employée pour fixer la relation entre l'effort total et la taille, Walston et Félix ont déterminé d'autres relations :

- la relation entre la documentation et la taille

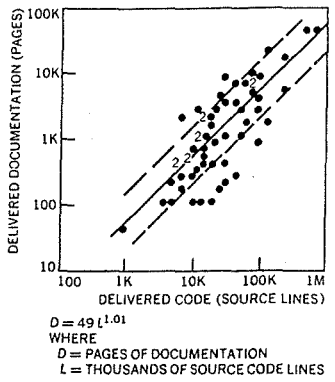


FIGURE 21 RELATION ENTRE LA DOCUMENTATION ET LA TAILLE

- la relation entre la durée du projet et sa taille

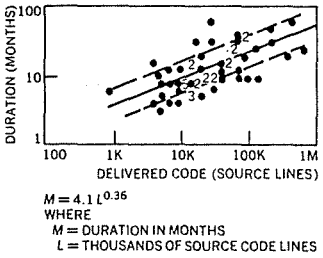


FIGURE 22 RELATION ENTRE LA DUREE ET LA TAILLE

- la relation entre la durée du projet et l'effort

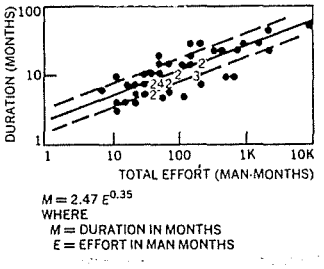


FIGURE 23 RELATION ENTRE LA DUREE ET L' EFFORT

- la relation entre les coûts hardware et la taille

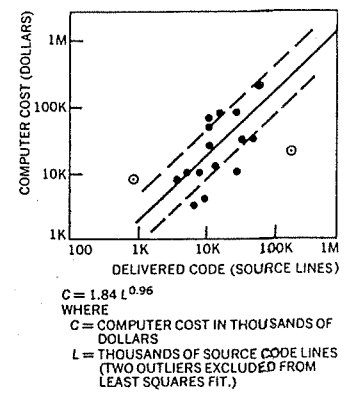


FIGURE 24 RELATION ENTRE LES COUTS HARDWARE ET LA TAILLE

- la relation entre la taille de l'équipe et l'effort

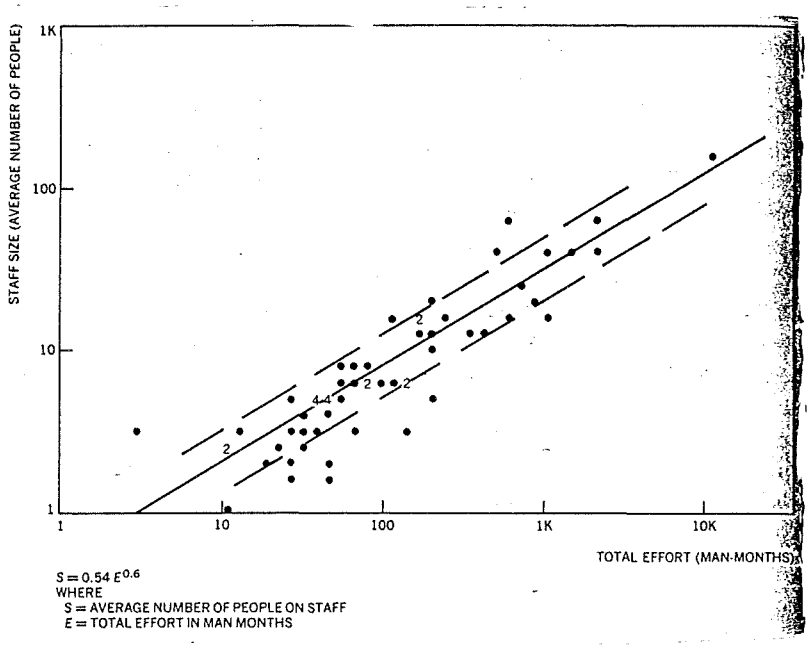


FIGURE 25 RELATION ENTRE LA TAILLE DE L'EQUIPE ET L'EFFORT

3. La méthode d'Halstead. [Halstead 77]

3.1. Motivations.

Halstead s'est consacré à l'étude d'une "science du logiciel" qu'il définit de la manière suivante : la "science logicielle" est un domaine des sciences naturelles récemment découvert qui, encore actuellement, est en pleine expansion. Au départ, elle se préoccupe des algorithmes et de leur implémentation, soit comme des programmes informatiques soit comme instruments de communication humaine. Comme une science expérimentale, elle traite seulement d'algorithmes qui peuvent être mesurés directement ou indirectement, statiquement ou dynamiquement; elle traite également des relations entre ces propriétés qui restent invariantes d'un langage à l'autre. Les investigations qui menèrent à la découverte de cette branche de la science physique furent motivées par le besoin reconnu d'une science informatique en général et d'un génie logiciel en particulier, pour obtenir une base importante d'une science théorique et expérimentale.

Dans une certaine mesure, ce besoin a été rencontré, par dérivation et validation expérimentale des relations qui semblent gouverner l'implémentation d'algorithmes quant à leur taille, leur niveau de langage, leur modularité, leur fiabilité, leur volume, leur contenu informationnel et le temps nécessaire pour les écrire. Leur approche est plutôt basée sur la validation expérimentale des relations que sur la démonstration de théorèmes. Par conséquent, comme dans chaque branche des sciences naturelles, chaque théorie développée ou chaque loi découverte est seulement valable dans l'environnement dans lequel elle a été testée [Halstead 77].

Au départ, comme Halstead l'expose, son but n'était pas de trouver une méthode d'estimation, mais de poser des jalons pour réaliser une "science du logiciel". Pour lui, les méthodes d'estimation ne sont qu'une application de cette théorie du logiciel.

De plus, la théorie d'Halstead est originale car elle propose une unité de mesure de la taille différente du LOC car celle-ci est soumise à de nombreuses interprétations.

3.2. Explication du modèle.

Halstead propose une nouvelle approche pour mesurer le code d'un programme. Il distingue dans une LOC les opérateurs des opérandes.

Les opérandes sont les constantes ou les variables contenues dans le code. Les opérateurs sont constitués des codes d'opérations, des délimiteurs, des symboles arithmétiques, de la ponctuation, etc... qui opèrent sur ou avec les opérandes. Les mots réservés tels que "do while", "if then", etc... qui contrôlent la séquence des opérations sont aussi des opérateurs.

Halstead définit six mesures de base que l'on retrouvera dans les différentes relations :

n_1 : le nombre d'opérateurs distincts apparaissant dans le code

n_2 : le nombre d'opérandes distinctes apparaissant dans le code

N_1 : le nombre de fois que tous les opérateurs sont utilisés

N_2 : le nombre de fois que toutes les opérandes sont utilisées

f_{1j} : le nombre d'occurrences du j*** opérateur dans le code pour $j = 1,\ldots,n_1$

f_{2j} : le nombre d'occurrences de la j*** opérande dans le code pour $j = 1,\ldots,n_2$

De ces mesures de base, on peut définir n, comme étant le vocabulaire :

$$n = n_1 + n_2$$

Le vocabulaire d'un programme est défini comme la somme des opérateurs et opérandes distincts utilisés dans le programme, et est la mesure du répertoire d'éléments qu'un programmeur doit manipuler pour implémenter un programme.

On peut également définir la longueur du programme (N) comme :

$$N = N_1 + N_2$$

La longueur d'un programme donné est définie comme la somme des opérateurs et opérandes utilisés. Intuitivement, la longueur est la mesure de la taille du programme, et mesure le nombre de fois qu'un programme manipule chacun des éléments de programmation.

A partir de ces définitions, il est utile de noter les trois relations suivantes :

$$N_1 = \sum_{1}^{n_1} f_{1j}$$

$$N_2 = \sum_{1}^{n_2} f_{2j}$$

$$N = \sum_{i}^{2} \sum_{j}^{n_i} f_{ij}$$

Exemple :l'algorithme d'Euclide [Halstead 77] écrit en Algol pour trouver le plus grand commun diviseur de deux nombres A et B.

```
        IF (a = 0)
LAST :  BEGIN GCD := B; RETURN END;
        IF (B = 0)
        BEGIN GCD := A; RETURN END;
HERE :  G := A/B; R := A-B*G;
        IF (R = 0) GO TO LAST;
        A := B; B := R; GO TO HERE
```

tableau des opérateurs

OPERATEURS	j	f_{1j}
;	1	9
:=	2	6
() ou begin.end	3	5
if	4	3
=	5	3
/	6	1
-	7	1
*	8	1
go to here	9	1
go to last	10	1
TOTAUX	n_1 =10	N_1 =31

tableau des opérandes

OPERANDES	j	f_{2j}
B	1	6
A	2	5
0	3	3
R	4	3
G	5	2
GCD	6	2
TOTAUX	n_2 =6	N_2 =21

Le fait qu'un algorithme soit uniquement constitué d'opérateurs et d'opérandes s'inspire du format d'instruction d'un ordinateur : le code d'opération et l'adresse d'opérande.

Remarque : de toute la théorie d'Halstead, nous ne retiendrons que les propriétés qui interviennent dans la détermination de la taille et de l'effort.

3.2.1. La relation entre longueur et vocabulaire

Halstead a suggéré une relation qui permet d'estimer la longueur d'un programme ($\hat{N}$) à partir du vocabulaire :

$$\hat{N} = n_1 \log_2 n_1 + n_2 \log_2 n_2$$

Le "^" signifie qu'il s'agit d'une quantité estimée et pas observée directement. Cependant, on peut rencontrer de grandes différences entre la longueur estimée et observée, mais pour une grande population de programmes, il existe une corrélation raisonnable (cfr. tableau 15) [Christensens, Fitsos, Smith 81].

Language	*Number of programs*	*Correlation coefficient*	*Cited reference*
FORTRAN	429	0.95	1
PL/I	120	0.98	8
COBOL	264	0.90+	6
System/370 assembly language	994	0.90+	4
PL/S	643	0.90+	4
COBOL	24	0.92	9
APL	29	0.96	9
RPG	371	0.94	10

TABLEAU 15 CORRELATION ENTRE $\hat{N}$ ET N

Etant donné cette corrélation, Halstead en dégage une première loi :

Loi 1 la longueur d'un programme est fonction du vocabulaire.

3.2.2. La taille du programme.

Bien que la longueur d'un programme puisse être considérée comme une mesure de la taille d'un programme, Halstead suggère une autre approche qui considère le nombre de fois (N) que les éléments du répertoire (n) sont utilisés dans le programme. Cette notion est exprimée comme le volume (V) d'un programme :

$$V = N \log_2 n$$

On peut remarquer que $\log_2 n$ est la longueur minimale en bits de tous les éléments individuels d'un programme. Le fait de multiplier $\log_2 n$ par N, c'est-à-dire le nombre de fois qu'apparaissent les éléments, permet d'obtenir la longueur totale, en bits, du programme. Cela revient à obtenir la longueur du code objet. Halstead en dégage une seconde loi :

Loi 2 LOC, longueur et volume sont trois mesures valables de la taille d'un programme.

En combinant les lois 1 et 2, on obtient la loi suivante :

Loi 3 la taille d'un programme mesurée dans un des trois termes, est une fonction du vocabulaire et donc, du langage utilisé.

3.2.3. Volume potentiel (V*)

Halstead suggère que la forme la plus succincte dans laquelle un algorithme pourrait être estimé, requiert l'usage d'un langage permettant de faire un appel à d'autres routines déjà définies et implémentées qui réalisent les opérations voulues (notion du "call"). Dans un tel langage, l'implémentation d'un algorithme ne requiert rien de plus que les noms des opérandes (arguments, résultats) [Mohanty 79].

Le volume potentiel ou encore le volume le plus petit possible est défini comme suit :

$$V^* = N^* \log_2 n^*$$

ou encore :

$$V^* = (N_1^* + N_2^*) \log_2 (n_1^* + n_2^*)$$

Maintenant, dans sa forme minimale, ni les opérateurs ni les opérandes n'apparaissent pas plus d'une fois dans le code. Donc, il vient que

$$N_1^* = n_1^* \quad \text{et} \quad N_2^* = n_2^*$$

Ceci nous donne :

$$V^* = (n_1^* + n_2^*) \log_2 (n_1^* + n_2^*)$$

De plus, le nombre minimal possible d'opérateurs n_1^*, pour chaque algorithme, est connu. Cela doit consister en un opérateur pour le nom de la procédure, et un autre qui sert

d'assignation et de symbole de groupement. Il vient que $n_1^* = 2$. Ce qui implique :

$$V^* = (2 + n_2^*) \log_2 (2 + n_2^*)$$

où n_2^*, pour de petits algorithmes du moins représenterait le nombre de paramètres input/output différents. De plus, l'avantage de cette équation est son indépendance vis à vis de tout langage et, puisque n_2^* est évaluée comme le nombre d'opérandes impliquées conceptuellement, le volume potentiel (V^*) semble être une bonne mesure du contenu informationnel de l'algorithme.

3.2.4. L'effort (E).

Après avoir défini le volume potentiel d'un programme, Halstead définit le niveau d'implémentation d'un programme (L) :

$$L = V^* / V$$

Halstead n'explicite pas cette notion, cependant, elle intervient dans la formule de l'effort de programmation (E) :

$$E = V / L = V^2 / V^*$$

D'après ce que nous avons pu comprendre de la théorie et selon V.R. Basili, le volume (V) représenterait le nombre de comparaisons mentales nécessaires pour générer un programme. En effet, supposons que l'implémentation d'un algorithme consiste en N sélections parmi un vocabulaire de n éléments et que la sélection est non aléatoire, c'est-à-dire réalisée avec un ordre de recherche binaire (impliquant $\log_2 n$ comparaisons pour la sélection de chaque élément), l'effort nécessaire pour générer un programme sera de $N \log_2 n$ comparaisons mentales. Ceci équivaut au volume du programme où chaque comparaison mentale requiert un nombre de discriminations mentales élémentaires qui représente une mesure de la difficulté (D) de la tâche. Halstead définit cette difficulté comme l'inverse du niveau de langage :

$$D = 1 / L$$

Il en résulte que :

$$E = V / L = V * D$$

Selon Basili, E est le plus souvent utilisé pour mesurer l'effort requis pour comprendre une implémentation plutôt que pour la produire; E est donc une mesure de la clarté du programme. En effet, Boehm fait souvent référence à la théorie d'Halstead quand il parle de la mesure de complexité d'un programme.

3.2.5. Le temps de developpement (T).

Pour calculer le temps de développement, la "science du logiciel" utilise le concept de "moment", défini par le psychologue Stroud comme le temps requis par le cerveau humain pour réaliser les discriminations les plus élémentaires. Ces "moments" surviennent à un taux variant de 5 à 20 par seconde. De ceci, Halstead définit la durée de développement du code d'un programme par :

$$T = E / S$$

où S désigne ces "moments". Halstead a fixé, de manière empirique, S à 18 pour son environnement.

4. La méthode de L.H. Putnam [Putman 79]

4.1. Motivations.

Putnam a remarqué que traditionnellement les managers faisaient deux hypothèses incorrectes concernant le développement d'un logiciel :

- le personnel et le temps sont interchangeables
- le niveau de productivité est relativement constant quel que soit le type d'applications au sein d'une même organisation.

La première hypothèse dit que l'effort de développement est simplement le produit du personnel et du temps, et que le temps peut être déterminé de manière arbitraire par le management. Le nombre de personnes nécessaires sera donc obtenu en divisant l'effort de développement prédéterminé (cfr figure 26 où l'on fait l'hypothèse que 50 personnes pendant 2 ans est équivalent à 33 personnes pendant 3 ans pour le même projet).

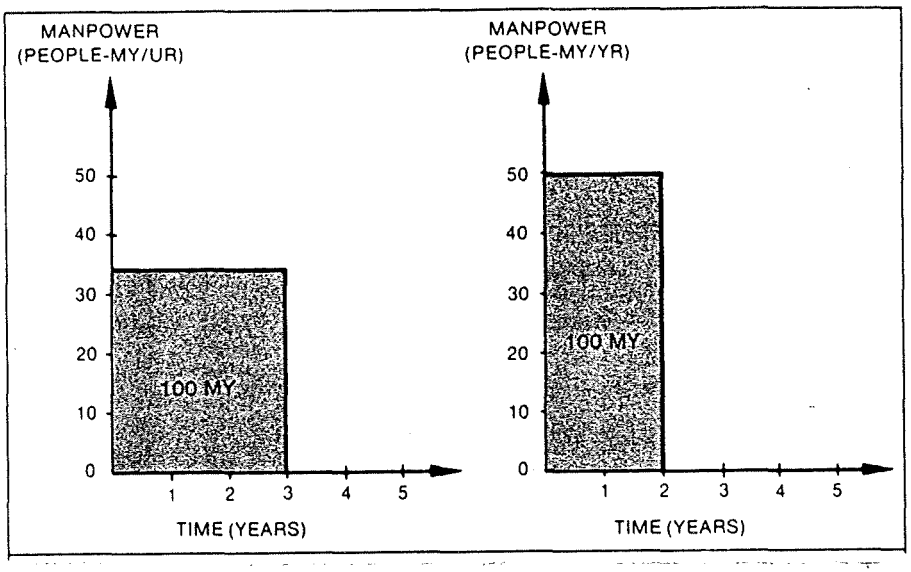


FIGURE 26

La seconde hypothèse avance que les chiffres de productivité obtenus pour d'anciens projets restent similaires pour de nouveaux projets. Cependant, ils n'examinent pas les caractéristiques précises de cette similarité. Une estimation totale des LOC dérivée des spécifications du projet est divisée par le taux de productivité issu d'anciens projets afin d'obtenir l'effort requis exprimé en année-homme (A-H).

Or, selon les études de Brooks, ces deux hypothèses sont erronées parce que la productivité n'est pas constante mais plutôt une fonction complexe de l'effort et des outils technologiques impliqués dans la tâche de développement, et parce que ajouter du personnel à un projet en retard ne fait qu'accroître ce retard.

Putnam, quant à lui, a étudié le comportement d'un projet tout au long de son développement afin de distribuer de manière optimale le personnel dans les différentes phases du cycle de vie d'un projet. Pour ce faire, il procéda en plusieurs étapes : il essaya tout d'abord d'estimer la taille d'un projet, ensuite de la convertir en une estimation du temps et de l'effort, et enfin de réaliser la meilleure distribution du personnel dans le développement.

4.2. Explication du modèle.

Pendant cinq ans, Putnam a étudié le nombre de personnes nécessaires à tout moment du développement pour une centaine de projets différents par leur type et leur taille. Tous ces projets ont montré un même modèle du cycle de vie, illustré par la courbe de la figure 27, c'est-à- dire une augmentation du

nombre de personnes jusqu'à atteindre un maximum (sommet) et ensuite une diminution des personnes impliquées dans le projet.

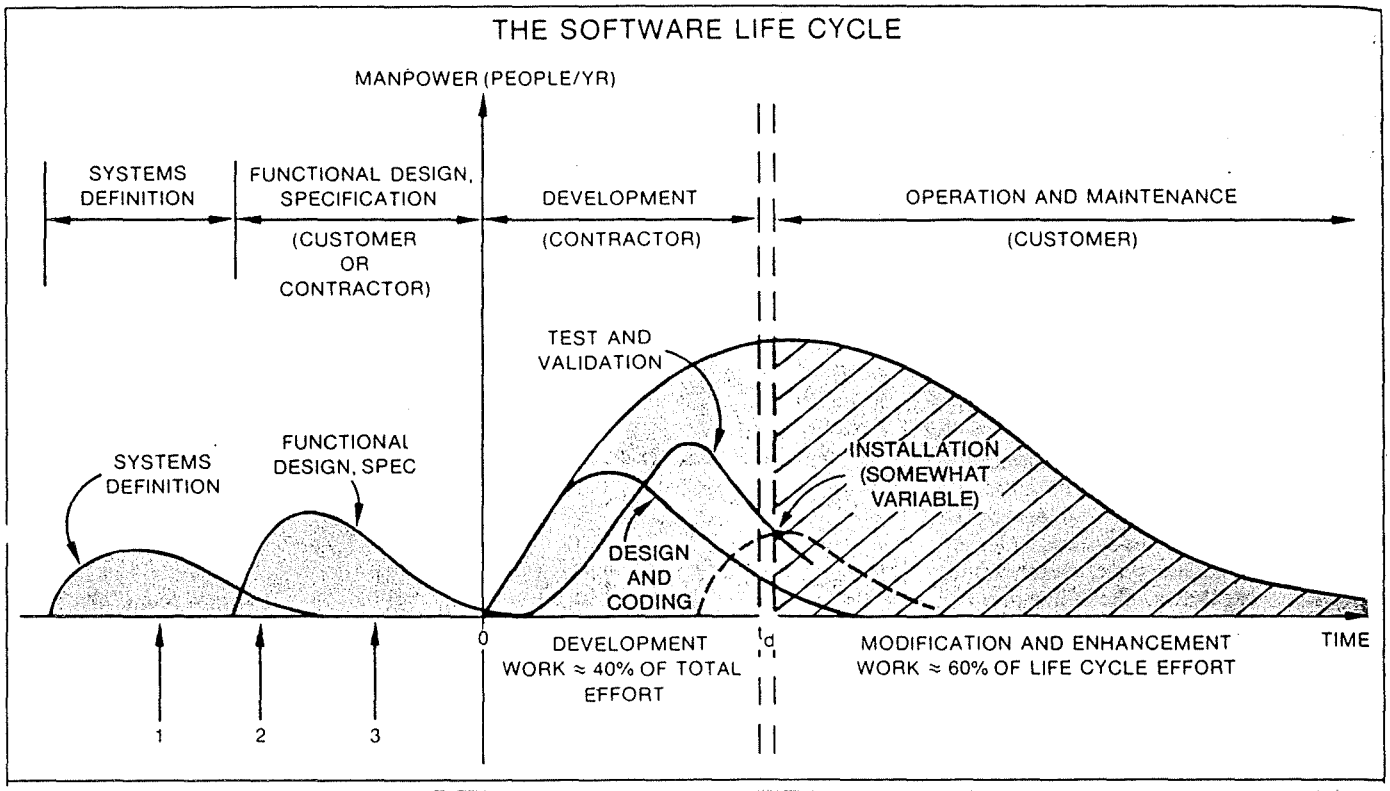


FIGURE 27 LE CYCLE DE VIE D'UN PROJET

L'usage de cette courbe et de l'équation correspondante lui permit de déterminer le nombre de personnes nécessaire à tout moment t :

$$Y = (K / (t_d)^2)\ t\ e^{-t^2 / 2t_d^2}$$

où Y = le nombre de personnes nécessaire au temps t

K = la surface en dessous de la courbe qui représente l'effort total de développement

t_d = le temps de développement, c'est-à-dire le moment où la courbe atteint son sommet, ce qui correspond à la fin de la phase de passage en production.

La portion décroissante de la courbe correspond à la phase de maintenance du projet. On peut dire que les grands logiciels suivent cette courbe appelée courbe de Rayleigh.

Afin de déterminer Y à un moment t, il faut connaître au préalable l'effort total K et le temps de développement K. On peut les connaître une fois la taille du système estimée à différentes étapes du développement; par exemple, au moment de la définition de projet et plusieurs fois durant les analyses conceptuelle et fonctionnelle (cfr figure 27). Putnam recommande d'estimer tout autant l'incertitude liée à l'estimation de la taille qui est à la base de son modèle et qui pourrait ce propager.

4.2.1. L'estimation de la taille.

4.2.1.1. Dans la définition de projet.

Pendant la première phase de la définition du système, il est nécessaire d'estimer la taille finale du projet; cependant, les données dont on dispose ne sont pas très nombreuses et très précises. Par conséquent, il paraît préférable d'estimer la taille en donnant un intervalle de grandeurs, ceci se faisant par analogie avec des projets précédents et par expérience.

Si a est la borne inférieure de l'intervalle et b la borne supérieure, c'est-à-dire le plus petit nombre et le plus grand nombre de LOC, on pourra déterminer la taille attendue et sa déviation standard en utilisant les lois des statistiques et des probabilités :

$$LOC = (a + b)/2$$

Et la déviation standard vaut :

$$\sigma_{Loc} = |b - a| / 6$$

C'est le mieux que l'on puisse faire à ce stade du développement.

Exemple: pour un système de taille importante, compte tenu de ce que l'on en connaît et de l'expérience acquise, on peut estimer approximativement la taille entre 50 000 et 140 000 LOC.

$$(50\ 000 + 140\ 000)/2 = 95\ 000\ LOC$$

et $\sigma_{Loc} = 90\ 000/6 = 15\ 000$ LOC.

On peut donc dire que la taille attendue est de 95 000 LOC à 15 000 LOC près. Par taille attendue, les statistiques entendent qu'il existe 68% de chance que la taille réelle se situe dans l'intervalle [80 000,110 000]. Il existe 99% de chance qu'elle se situe entre 50 000 et 140 000 LOC, et 1% de chance qu'elle soit en dehors de l'intervalle.

4.2.1.2. Dans l'analyse conceptuelle.

Si on continue dans le cycle de vie, on peut réduire le risque d'erreur si l'on divise le projet en plusieurs parties à estimer séparément. Cette division est possible car l'information disponible se précise. Pour chaque partie, on reprend les mêmes équations d'estimation. Il suffira ensuite de rassembler les résultats obtenus pour produire une estimation globale.

4.2.1.3. Dans l'analyse fonctionnelle.

Au début de l'analyse fonctionnelle, on connaît les sous-systèmes. A ce moment, les membres de l'équipe qui ont travaillé à la définition du système estiment la taille de chaque sous-système de la manière suivante :

a = la plus petite taille possible du sous-système

m = la taille la plus vraisemblable

b = la plus grande taille possible du sous-système.

La valeur attendue (LOC_i) pour la taille du sous-système i égale :

$$LOC_i = (a + 4m + b)/6$$

Et la déviation standard vaut :

$$\sigma_{LOC_i} = |b - a| / 6$$

Généralement, les gens préfèrent estimer en donnant un intervalle de grandeurs plutôt qu'une simple estimation. En effet, de cette manière, ils peuvent donner un intervalle plus large ou plus étroit selon les données dont ils disposent. De plus, psychologiquement, cela les engage moins.

La taille totale du système sera obtenue par :

$$LOC = \sum_{1}^{n} LOC_i$$

$$\sigma_{LOC} = \left(\sum_{i} \sigma_{LOC}^2\right)^{\frac{1}{2}}$$

Si on reprend le même exemple et qu'on en fait trois sous-systèmes :

SOUS-SYSTEME	a	m	b	LOC_i	σ_{LOC_i}
1	25000	40000	70000	42500	7500
2	5000	15000	26000	15167	3500
3	12000	36000	50000	34333	6333
			TOTAL :	92000	10422

Maintenant, la taille attendue est de 92 000 LOC à 10 422 LOC près, avec 68% de chance d'être dans l'intervalle [81578,102422] et 99% de chance d'être dans l'intervalle [60735,823264]. On peut remarquer que l'incertitude est ainsi réduite de 15 000 à 10 422 LOC, simplement par une division du système en trois parties.

Au fur et à mesure que l'on avance dans l'analyse fonctionnelle, de nouveaux sous-systèmes se définiront et l'estimation en sera d'autant plus précise. Par ce genre de démarche, l'incertitude (la déviation standard) peut diminuer de moitié.

4.2.2. La conversion de l'estimation de la taille en une estimation du temps, de l'effort et du coût.

Il existe une relation importante dans le développement du logiciel entre le nombre de LOC et de l'effort, le temps de développement et l'état de la technologie qui est appliquée pour le projet. L'équation décrivant cette relation est :

$$LOC = C_K \; K^{1/3} \; t_d^{4/3}$$

où K = l'effort en A-H
t_d = le temps de développement
C_K = la constante liée à l'état de la technologie.

4.2.2.1. La détermination du temps et de l'effort.

Putnam par observation, a déterminé un graphe dans lequel il met en relation la taille du projet avec le temps de développement et l'effort nécessaire pour le développer (cfr figure 28).

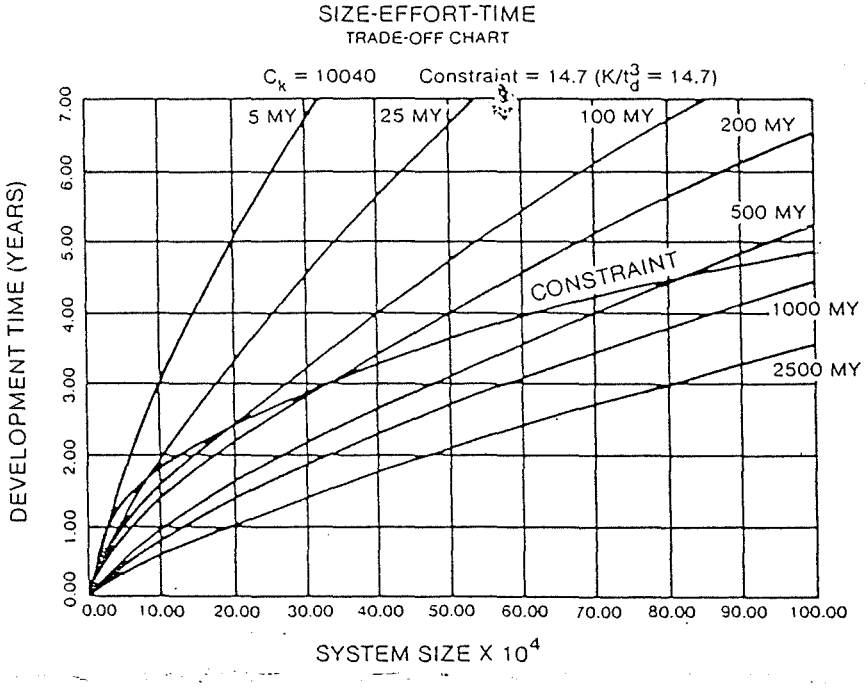


FIGURE 28 TAILLE - EFFORT - TEMPS

Par exemple, pour un projet de 200 000 LOC, le temps de développement minimal sera égal à 2.5 années et requiert un effort de 100 A-H.

On peut également remarquer que Putnam a déterminé un temps minimal, en fonction de la taille du projet, en dessous duquel aucun développement n'est réalisable. C'est illustré sur le graphe par la courbe de contrainte ayant pour équation

$$K / t_d^3 = 14.7$$

Ainsi, il est impossible de développer un projet de 200 000 LOC en moins de 2.5 années. Ces valeurs données dans le graphe différeront d'une organisation à l'autre et pour des types d'applications différents au sein d'une même organisation. Ces valeurs sont déterminées d'une part par la valeur attribuée à la constante de technologie C_K et, d'autre part, par l'utilisation ou non de techniques de programmation modernes, par le langage utilisé, par l'environnement de développement (on line, interactif, batch,...) et aussi par la disponibilité de la machine. On remarquera cependant que la détermination des valeurs de ces composantes et leur importance relative ne sont pas faciles à comprendre et à mesurer.

4.2.2.2 La détermination du coût.

Les coûts de développement sont surtout des coûts liés au personnel. Putnam prétend que les autres coûts tels que ceux liés au hardware par exemple, sont relativement constants dans une organisation et sont par conséquent connus. Il est donc possible de savoir quel est le coût d'une A-H. Par conséquent, le coût de développement se calcule facilement en multipliant le nombre d'A-H par le coût d'une A-H.

Remarques :

- il faut être conscient que toutes ces estimations reposent sur des informations comprenant un certain degré d'incertitude. Par conséquent, il faut être prudent et prendre chaque résultat avec circonspection. Il faut toujours avoir à l'esprit l'importance du risque.

- Putnam recommande également de recourir à la programmation linéaire; celle-ci est la meilleure technique qui produit la meilleure solution possible à un problème soumis à un certain nombre de contraintes (de management par exemple). Dans le cas particulier qui nous occupe, on pourrait adopter cette technique pour minimiser, soit le temps, soit les coûts ou encore appliquer deux fois la technique pour chacune de ces deux contraintes. La solution qui minimise le temps de développement est aussi la solution qui requiert le plus long temps de développement tandis que la solution qui minimise le temps de développement (sans pour autant tomber en dessous de la courbe de contrainte) est la solution la plus coûteuse. Entre ces deux extrêmes, se situe la région de toutes les autres solutions possibles.

4.2.3. La détermination de la distribution optimale des personnes à tout moment du développement.

Etant donné que nous avons pu estimer l'effort et le temps de développement en fonction de la taille du projet, il ne reste plus qu'à en déduire quelle sera la charge en hommes à chaque instant du cycle de vie, c'est-à-dire appliquer la formule :

$$Y = (K / (t_d)^2)\ t\ e^{-t^2 / 2t_d^2}$$

La courbe générale de distribution des personnes tout au long des phases du développement a la forme présentée à la figure 29.

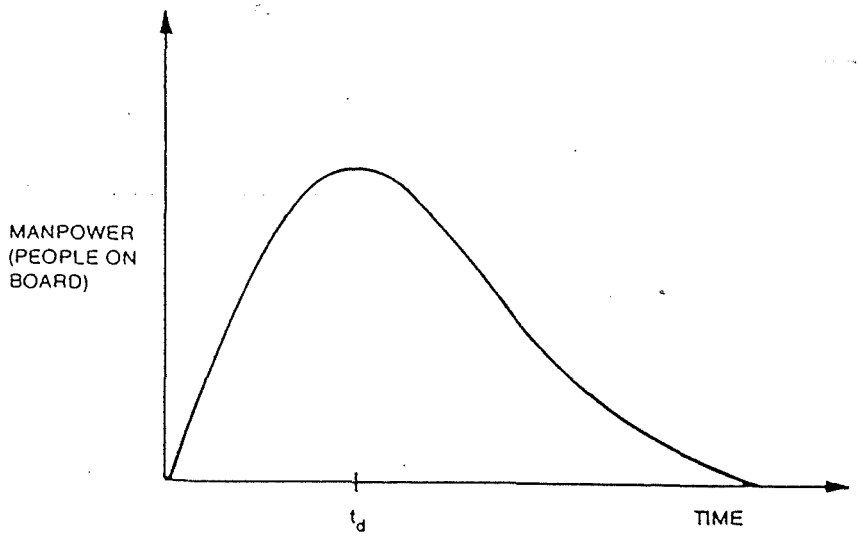


FIGURE 29 COURBE DE LA FORCE DE TRAVAIL

Si on suppose que l'effort total est estimé à 100 A-H pour un projet donné, la figure 30 montre la distribution de cet effort en faisant varier la valeur de t_d en vue de respecter des échéances de livraison ou tout autre contrainte. On peut voir ainsi l'effet d'une compression du temps de développement ou au contraire l'effet d'un prolongement sur la distribution des personnes.

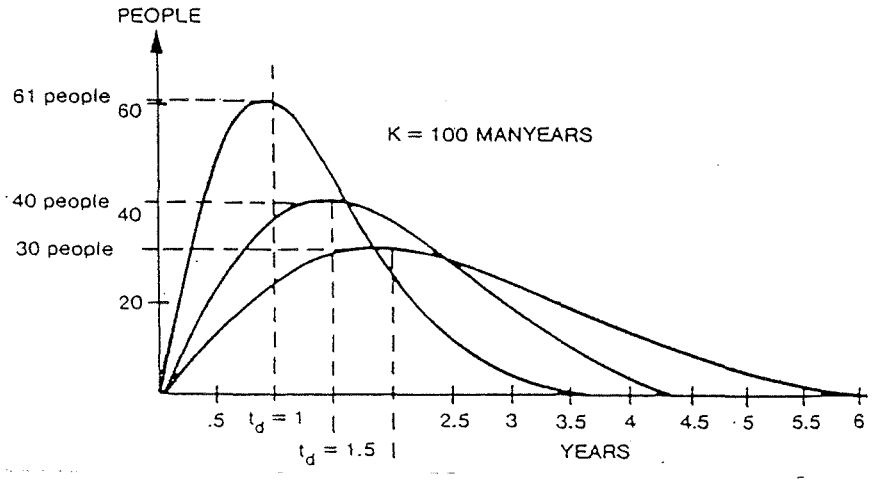


FIGURE 30 DIFFERENTES COURBES DE LA FORCE DE TRAVAIL

Mais de toute façon, il existe un temps de développement minimal en dessous duquel il est impossible de développer un

projet même avec une armée de programmeurs. En fait, ce temps de développement minimal corrobore l'étude de Brooks sur la non interchangeabilité entre le temps et le personnel.

5. La méthode de Boehm (le modèle cocomo). [Boehm 81]

5.1. Motivations.

Le premier but de Boehm était sûrement la réalisation d'une étude complète sur l'aspect économique du développement de logiciels. En effet, le titre de son livre le confirme : "Software engineering economics".

Non seulement il a étudié des méthodes d'estimation de coût de développement mais aussi il a tenté d'appliquer des techniques propres à l'économie pour analyser l'aspect économique du logiciel :

- l'analyse coût - efficacité,
- l'économie d'échelle, le rendement,
- l'analyse de sensibilité,
- les ratios...

Boehm a été très complet dans ses études et a abordé de nombreux sujets relatifs aux caractéristiques économiques du génie logiciel, mais en ce qui nous concerne, nous nous limiterons à essayer de dégager ses motivations relatives aux méthodes d'estimation : il voulait aider les gens à comprendre et à déterminer les coûts d'un logiciel. Pour ce faire, il réalisa une étude très précise pour déterminer les éléments constitutifs du coût, c'est-à-dire les facteurs de coût. L'originalité de son modèle est d'avoir réparti le coût total d'un logiciel sur chacune des phases du développement en discernant l'influence relative d'un facteur sur chaque phase.

5.2. Explication du modèle.

Cocomo est constitué d'une hiérarchie de trois modèles : le modèle de base, le modèle intermédiaire et le modèle détaillé. Ceux-ci fournissent des estimations allant de l'estimation de l'effort brute à une estimation de l'effort plus précise.

Par estimation brute de l'effort (macro-estimation), on entend une estimation de l'effort calculée seulement en fonction de la taille du projet.

Par contre, une estimation plus précise (micro- estimation) est l'estimation d'un sous-système ou d'un module obtenue par division d'un projet. Ici encore, on corrige une estimation brute en tenant compte de l'influence de plusieurs facteurs pour l'effort requis à chacune des phases du développement.

5.2.1. Le modèle de base (macro-estimation).

L'entrée de ce modèle est la taille du projet en LOC obtenue soit à priori par estimation, soit à postériori par observation. En fonction de cette taille, on obtient l'effort requis pour développer le projet selon la formule générale :

$$E = a\ KLOC^b$$

Différents modes de développement sont distingués et déterminent les valeurs des constantes a et b :

- le mode "organic" où a = 2.4 et b = 1.05,
- le mode "semi-détaché" où a = 3 et b = 1.12,
- le mode "imbriqué" où a = 3.6 et b = 1.20.

Le mode "organic" : dans ce mode, on développe des projets de taille relativement petite dans un environnement familier. Les développeurs ont une grande expérience du domaine d'application. Le temps de communication entre les personnes de l'équipe sont très réduits.

Le mode "semi-détaché" : il représente un mode intermédiaire entre le mode "organic" et le mode "imbriqué".

Le mode "imbriqué" : la principale caractéristique de ce mode de développement est la nécessité de faire face à des contraintes de développement très sévères. Beaucoup d'effort est dépensé pour assurer une correspondance entre les spécifications et le produit final car le domaine n'est pas familier. Il en résulte une productivité basse, des équipes petites et un temps de communication très élevé.

Ici Boehm ne fait pas preuve de beaucoup de précision car, nulle part dans son étude, il n'explique précisément comment il est arrivé à ce type de formule et la manière dont il a fixé les valeurs de a et b en fonction du mode de développement. Il fallait le souligner.

5.2.2. Le modèle intermédiaire.

Il estime le coût d'un logiciel de la façon suivante :

Première étape : une macro-estimation (cfr le modèle de base).

Deuxième étape : on détermine un ensemble de multiplicateurs d'effort pour 15 facteurs de coût.

Troisième étape : l'effort de développement est estimé en multipliant l'effort nominal (calculé en première étape) par les multiplicateurs d'effort.

Quatrième étape : des calculs additionnels peuvent être réalisés pour déterminer, à partir de l'effort de développement, le coût en francs, le planning de développement, la distribution des activités, le coût hardware, le coût annuel de maintenance, etc...

5.2.2.1. Première étape : l'estimation de l'effort nominal selon le mode de développement.

Soit l'estimation du coût de développement d'un micro-processeur pour le transfert de fonds électronique avec un haut degré de fiabilité, de performance et d'interfaçage. Etant donné ces caractéristiques, il s'agit d'un projet à développer dans le mode "imbriqué". On estime sa taille à 10 000 LOC et l'effort de développement nominal est égal à

$$E = 2.8 * 10^{1.2} = 44 \text{ M-H.}$$

5.2.2.2. Deuxième étape : les multiplicateurs d'effort.

Chacun des 15 facteurs de coût a une échelle de complexité (très basse, basse, nominale, haute, très haute) et un ensemble de multiplicateurs d'effort qui indique dans quelle mesure un effort nominal estimé (à la première étape) doit être multiplié pour tenir compte du niveau de complexité de l'attribut caractérisant le projet (cfr tableau 16).

	Ratings					
Cost Drivers	Very Low	Low	Nominal	High	Very High	Extra High
Product Attributes						
RELY Required software reliability	.75	.88	1.00	1.15	1.40	
DATA Data base size		.94	1.00	1.08	1.16	
CPLX Product complexity	.70	.85	1.00	1.15	1.30	1.65
Computer Attributes						
TIME Execution time constraint			1.00	1.11	1.30	1.66
STOR Main storage constraint			1.00	1.06	1.21	1.56
VIRT Virtual machine volatility*		.87	1.00	1.15	1.30	
TURN Computer turnaround time		.87	1.00	1.07	1.15	
Personnel Attributes						
ACAP Analyst capability	1.46	1.19	1.00	.86	.71	
AEXP Applications experience	1.29	1.13	1.00	.91	.82	
PCAP Programmer capability	1.42	1.17	1.00	.86	.70	
VEXP Virtual machine experience*	1.21	1.10	1.00	.90		
LEXP Programming language experience	1.14	1.07	1.00	.95		
Project Attributes						
MODP Use of modern programming practices	1.24	1.10	1.00	.91	.82	
TOOL Use of software tools	1.24	1.10	1.00	.91	.83	
SCED Required development schedule	1.23	1.08	1.00	1.04	1.10	

* For a given software product, the underlying virtual machine is the complex of hardware and software (OS, DBMS, etc.) it calls on to accomplish its tasks

TABLEAU 16 LES MULTIPLICATEURS D'EFFORT

Les résultats de cette étape sur l'exemple cités ci-dessus sont repris dans le tableau suivant :

Cost Driver	Situation	Rating	Effort Multiplier
RELY	Serious financial consequences of software faults	High	1.15
DATA	20,000 bytes	Low	0.94
CPLX	Communications processing	Very High	1.30
TIME	Will use 70% of available time	High	1.11
STOR	45K of 64K store (70%)	High	1.06
VIRT	Based on commercial microprocessor hardware	Nominal	1.00
TURN	Two-hour average turnaround time	Nominal	1.00
ACAP	Good senior analysts	High	0.86
AEXP	Three years	Nominal	1.00
PCAP	Good senior programmers	High	0.86
VEXP	Six months	Low	1.10
LEXP	Twelve months	Nominal	1.00
MODP	Most techniques in use over one year	High	0.91
TOOL	At basic minicomputer tool level	Low	1.10
SCED	Nine months	Nominal	1.00
	Effort adjustment factor (product of effort multipliers)		1.35

TABLEAU 17 RESULTATS POUR LE LOGICIEL DU MICRO-PROCESSEUR

L'effet d'une erreur dans le logiciel sur le système de transferts de fonds électroniques peut être une perte financière très importante. Dès lors, l'attribut "rely" du projet est haut et le multiplicateur d'effort pour réaliser un haut niveau de fiabilité est de 1.15 où 15% d'effort supplémentaire sont requis par rapport à une fiabilité nominale.

La détermination du niveau de complexité et du multiplicateur d'effort pour les autres attributs s'effectue de la même manière, par l'expérience, excepté pour le facteur "complexité" qui est obtenu via le tableau 18 :

Rating	Control Operations	Computational Operations	Device-dependent Operations	Data Management Operations
Very low	Straightline code with a few nonnested SP[a] operators: DOs, CASEs, IFTHENELSEs. Simple predicates	Evaluation of simple expressions: e.g., A = B + C * (D − E)	Simple read, write statements with simple formats	Simple arrays in main memory
Low	Straightforward nesting of SP operators. Mostly simple predicates	Evaluation of moderate-level expressions, e.g., D = SQRT (B**2-4.*A*C)	No cognizance needed of particular processor or I/O device characteristics. I/O done at GET/PUT level. No cognizance of overlap	Single file subsetting with no data structure changes, no edits, no intermediate files
Nominal	Mostly simple nesting. Some intermodule control. Decision tables	Use of standard math and statistical routines. Basic matrix/vector operations	I/O processing includes device selection, status checking and error processing	Multi-file input and single file output. Simple structural changes, simple edits
High	Highly nested SP operators with many compound predicates. Queue and stack control. Considerable intermodule control.	Basic numerical analysis: multivariate interpolation, ordinary differential equations. Basic truncation, roundoff concerns	Operations at physical I/O level (physical storage address translations; seeks, reads, etc). Optimized I/O overlap	Special purpose subroutines activated by data stream contents. Complex data restructuring at record level
Very high	Reentrant and recursive coding. Fixed-priority interrupt handling	Difficult but structured N.A.: near-singular matrix equations, partial differential equations	Routines for interrupt diagnosis, servicing, masking. Communication line handling	A generalized, parameter-driven file structuring routine. File building, command processing, search optimization
Extra high	Multiple resource scheduling with dynamically changing priorities. Microcode-level control	Difficult and unstructured N.A.: highly accurate analysis of noisy, stochastic data	Device timing-dependent coding, micro-programmed operations	Highly coupled, dynamic relational structures. Natural language data management

TABLEAU 18 TAUX DE COMPLEXITE VERSUS TYPE DE MODULE

On détermine d'abord que le système de communication est classifié comme une opération dépendant du matériel (colonne 4 du tableau 18); dans cette colonne, on détermine qu'une manipulation de ligne de communication a un niveau de complexité "very high". Du tableau 16, on dégage le multiplicateur d'effort correspondant qui est égal à 1.3.

5.2.2.3. Troisième étape : l'estimation de l'effort de développement.

On peut alors calculer l'effort de développement pour un logiciel de communication comme étant l'effort de développement nominal (44 M-H) multiplié par le produit des multiplicateurs

d'effort des 15 facteurs de coût (1.35 dans le tableau 17). Dès lors, l'effort estimé pour le projet est de : 44 M-H * 1.35 = 59 M-H.

5.2.2.4. Quatrième étape : les calculs additionnels.

Il existe une formule, par exemple, pour calculer le temps de développement en fonction de l'effort et du mode de développement. Pour le calculer, on emploie la formule :

$T_{d\bullet v} = 2.5\ (59)^{0.32} = 9$ mois, dans le mode imbriqué

On peut remarquer que $T_{d\bullet v}$ correspond au t_d du modèle de Putnam.

Remarque : par l'utilisation des tables où sont repris les différents niveaux de complexité pour les facteurs, il est possible de réaliser des analyses de sensibilité du temps de développement, suite à une variation du niveau de complexité.

5.2.3. Le modèle détaillé.

Les quatre étapes du modèle précédent sont encore suivies, non plus pour estimer le projet dans son entièreté mais pour l'estimer par parties, c'est-à-dire par sous-systèmes et/ou modules. Boehm a proposé une décomposition du type système - sous-systèmes - modules .

5.2.3.1. Première étape : la taille totale du système et l'effort nominal.

On estime les tailles des modules d'un sous-système que l'on somme pour obtenir la taille du sous-système. La taille totale du système sera la somme des tailles des sous-systèmes (approche bottom-up). A partir de cette taille et suivant le mode de développement, on peut calculer l'effort nominal du développement (cfr la première étape du modèle intermédiaire).

5.2.3.2. Deuxième étape : l'influence des facteurs aux différents niveaux de la décomposition.

Boehm a remarqué que les facteurs n'ont pas la même influence selon les trois niveaux de décomposition. Au niveau le plus bas (c'est-à-dire le module), le modèle est caractérisé par un certain nombre de facteurs tels que la complexité du module, l'adaptation d'un logiciel existant, la capacité du programmeur à réaliser le module, l'expérience du langage de programmation, etc... Ces facteurs ont une influence différente d'un module à l'autre, ils sont donc estimés séparément pour chaque module. Par contre, il existe un certain nombre de facteurs qui ont une influence similaire sur tous les modules d'un même sous-système : ce sont les contraintes de temps et de mémoire, la capacité des analystes, les outils, etc... Ces facteurs sont alors estimés au niveau du sous-système. Ces

considérations sont également valables pour le niveau sous-système; en effet, des sous-systèmes ont des caractéristiques similaires ou non.

Après avoir distingué les facteurs influant au niveau du module ou au niveau du sous-système, il est possible de corriger par des facteurs multiplicateurs l'estimation nominale de l'effort nécessaire pour développer un module ou un sous-système.

Dans le modèle détaillé, Boehm reprend les facteurs de coût du modèle intermédiaire mais de manière plus précise; il attribue à chacun d'entre eux une valeur d'influence selon la phase de développement considérée.

Exemple :pour le facteur "modern programming practices" et pour un niveau d'utilisation "very low", le multiplicateur vaut 1.1 pour la phase "detailed design" et 1.5 pour la phase "integration and tests" (cfr la figure 31 suivante).

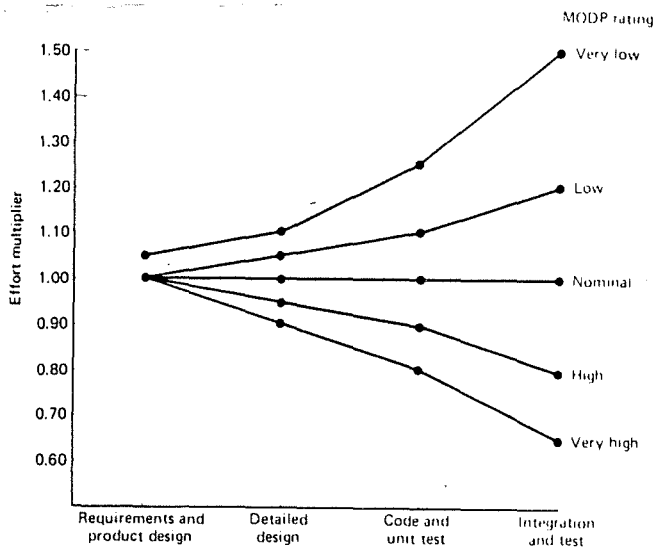


FIGURE 31 LES MULTIPLICATEURS D'EFFORT PAR PHASE POUR LE FACTEUR "UTILISATION DE TECHNIQUES MODERNES DE PROGRAMMATION"

On peut ainsi remarquer le niveau de précision qu'a atteint Boehm en estimant un projet par ses modules et sous-systèmes et en distinguant l'influence des facteurs de coût à travers les phases du cycle de vie d'un projet.

6. Conclusion.

Dans ce chapitre, nous avons décrit le fonctionnement des méthodes d'estimation les plus connues. On peut dire que, bien qu'elles soient différentes dans leur approche du problème, elles suivent les mêmes lignes directrices :

- elles reconnaissent qu'il existe une relation entre l'effort et la taille d'un projet

- la première estimation de l'effort est corrigée par des facteurs de coût liés à l'environnement, au personnel, au logiciel, etc...

Puisqu'elles respectent les mêmes principes, on pourraient s'attendre à ce qu'elles fournissent des estimations relativement équivalentes pour un même projet. Cependant, les études de Mohanty et Kemerer prouvent le contraire. En effet, Mohanty a estimé les coûts d'un logiciel hypothétique en utilisant 13 modèles différents; les coûts attendus variaient de 362500 $ à 2766667 $ et le temps de développement attendu variait de 13.77 à 25.8 mois malgré une taille et des coûts par instruction identiques dans les 13 modèles [Mohanty 81].

Kemerer, de même, a étudié 3 modèles (SLIM de Putnam, COCOMO de Boehm et les points de fonction d'Albrecht) pour l'estimation de 15 projets dont la plupart étaient écrits en COBOL et "tournaient" sur IBM. Par exemple pour un projet en particulier, SLIM estimait l'effort à 3857.8 M-H, le COCOMO de base à 1095.10 M-H, le COCOMO intermédiaire à 910.56 M-H, le COCOMO détaillé à 932.96 M-H et les points de fonction à 344 M-H [Kemerer 87].

Mohanty et Kemerer attribuent ces écarts à la base de données sur laquelle chaque modèle repose. En effet, celle- ci comprend les coefficients qui sont utilisés pour fixer les constantes présentes dans chaque modèle. Dès lors, les estimations s'écartent très nettement de la réalité; en effet, pour le projet décrit dans Kemerer, l'effort de développement réel était de 287 M-H et SLIM l'a estimé 3857.8 M-H (ce qui équivaut à un écart de 1 à 13).

Il ressort de l'étude de Kemerer que la méthode des points de fonction "colle" le mieux à la réalité car elle serait la plus indépendante par rapport à son environnement d'expérimentation.

CHAPITRE 6.

CONCLUSION GENERALE DE LA PREMIERE PARTIE.

L'objectif de cette première partie était de cerner la problématique relative aux méthodes d'estimation des coûts de développement de logiciel, afin de mieux situer parmi celles-ci la méthode des points de fonction.

Premièrement, nous pouvons constater qu'il n'existe pas à l'heure actuelle de modèle réalisant à priori l'estimation de l'effort, du temps de développement et des coûts. En effet, toutes les méthodes connues réalisent à postériori des mesures de la productivité des équipes de développement à partir d'une taille observée du logiciel pour en dégager l'effort de développement. Si on veut réaliser à priori une estimation à priori de la taille d'un logiciel, les méthodes ne proposent qu'une démarche par analogie, ou encore se fient à l'intuition et à l'expérience des estimateurs. Rien de précis et de bien rigoureux n'existe par conséquent.

Deuxièmement, nous pouvons dire que la méthode des points de fonction propose une démarche opposée à celle des méthodes classiques car, bien qu'elle fonctionne également à postériori, elle estime la taille d'un logiciel à partir des fonctionnalités demandées par l'utilisateur et non plus sur base d'une observation du nombre de LOC. C'est ici que la méthode présente une originalité puisque les fonctionnalités d'un logiciel sont connues avec un degré de précision plus ou moins grand, dès la définition de projet. Par conséquent, cette méthode ouvre une porte vers des estimations à priori.

De cette première partie, nous pouvons encore retirer plusieurs enseignements tels que :

- la difficulté de déterminer les facteurs de coût les plus pertinents et d'en connaître leur juste influence;
- les problèmes liés à l'aspect non quantitatif de certains facteurs;
- le fait qu'aucune méthode n'envisage l'influence sur les coûts de l'interdépendance des facteurs;
- le manque de consensus dans la terminologie employée;
- la dépendance de chaque méthode par rapport à son environnement d'expérimentation.
- le manque de clarté dans les définitions des concepts employés dans les modèles.

Nous pensons qu'il serait bon de terminer cette première partie en proposant, par l'intermédiaire de 13 critères, les qualités que devrait posséder toute méthode d'estimation. La majorité de ces critères est reprise de l'article de Boehm et Wolverton [Boehm, Wolverton 80]. Ces critères peuvent constituer non seulement une base pour l'évaluation de la qualité d'une méthode d'estimation mais aussi une base pour la comparaison des méthodes.

La définition : le modèle définit-il clairement les coûts à estimer et ceux à écarter de l'estimation ? Quel cycle de vie est adopté ?

La fiabilité : dans quelle mesure les estimations se rapprochent-elles du coût réel ?

L'objectivité : s'il existe un écart entre l'estimation et la réalité, dans quelle mesure cet écart provient-il de l'évaluation de facteurs subjectifs (difficiles à évaluer) intervenant dans la méthode ? Dans quelle mesure est-il possible d'influencer le modèle pour arriver à une estimation voulue ?

La compréhensibilité : l'utilisateur peut-il comprendre comment les résultats du modèle sont produits ?

Les détails : est-il possible d'appliquer le modèle à des sous-systèmes d'une application (tâches/activités) ?

La stabilité : une petite modification de l'entrée du modèle entraîne-t-elle une petite modification du résultat ?

Le domaine d'application : quelles classes d'application peuvent être estimées par ce modèle ? Le modèle est-il facilement adaptable à d'autres domaines d'application ?

La facilité d'utilisation : les entrées et les options du modèle sont-elles faciles à comprendre et à utiliser ?

L'aspect prévisionnel : à quelle étape du cycle de vie peut-on réaliser l'estimation ? Le modèle demande-t-il que les informations soient connues avec certitude avant que le projet ne soit terminé ?

La pertinence et la complétude du choix des facteurs : quels sont les facteurs de coût employés et sont-ils influants ?

La rapidité versus la qualité de l'estimation : quel effort doit-on consentir pour obtenir une estimation ? La qualité de cette estimation est-elle en rapport avec l'effort consenti ?

La possibilité d'améliorer l'estimation par l'expérience : la méthode permet-elle la correction des paramètres sur lesquels est basée son estimation ?

L'indépendance vis à vis de l'environnement d'expérimentation : dans quelle mesure peut-on utiliser la méthode dans un environnement différent de celui dans lequel elle a été expérimentée ?

Partie 2

La méthode des points de fonction.

CHAPITRE 1.

LES BASES DE LA METHODE.

1. Les motivations.

Dans le cadre du symposium tenu en Californie par les utilisateurs IBM (Guide Share 1979), Albrecht proposa la technique des points de fonction afin de calculer le gain de productivité d'une équipe de développement suite à l'adoption de nouvelles techniques et langages de programmation. On peut voir sur le graphique suivant le résultat de son enquête :

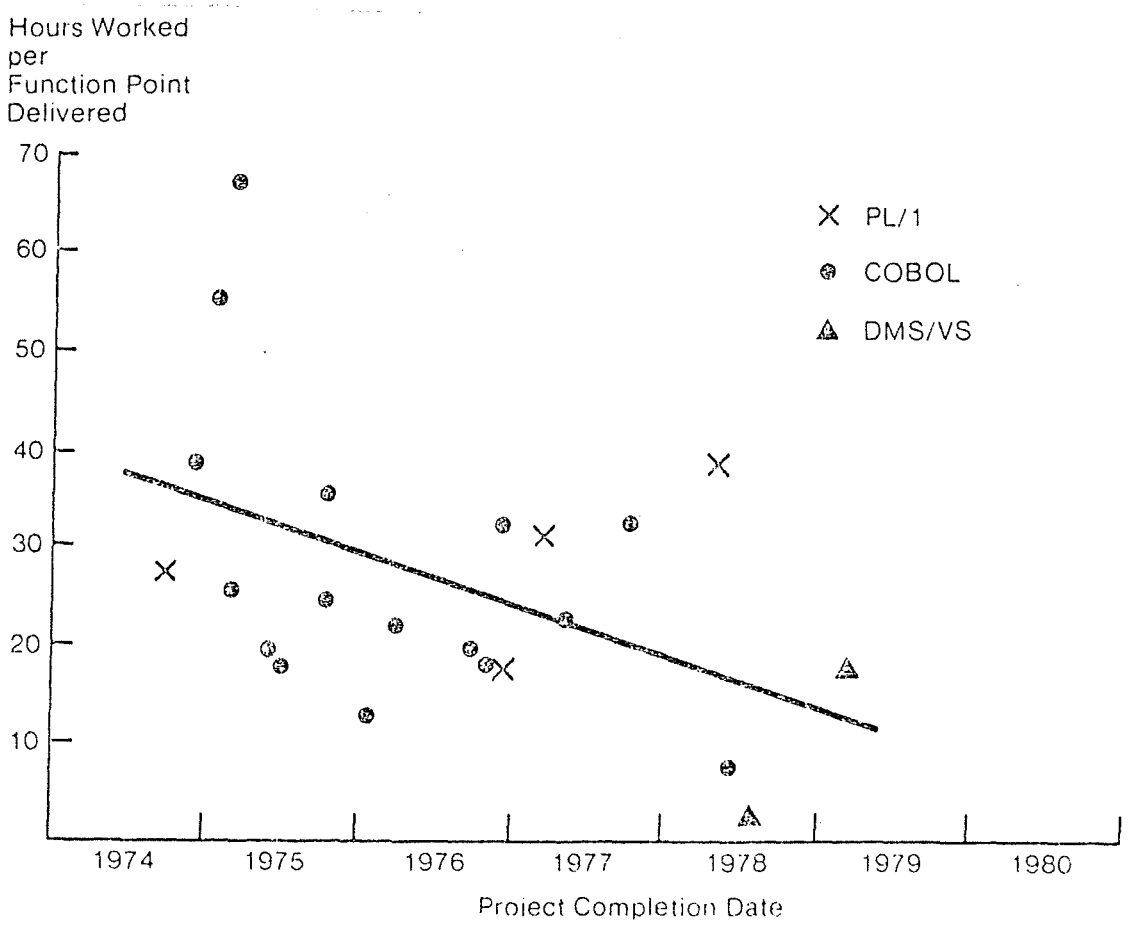


FIGURE 3.2 L'EVOLUTION DE LA PRODUCTIVITE DANS LE SERVICE DE DEVELOPPEMENT D'IBM

Pour ce faire, il a dû comparer la productivité du développement de projets écrits dans des langages différents, en mesurant le produit et l'effort nécessaire pour les réaliser.

Quant à la mesure de l'effort, Albrecht se contente d'enregistrer au fur et à mesure le temps des développeurs et du client consacré au développement. A la fin du développement, il connaît ainsi l'effort total.

Quant à la mesure du produit, Albrecht ne pouvait pas utiliser la mesure classique de la taille exprimée en LOC, car celle-ci était dépendante du langage utilisé; dès lors, toute comparaison était impossible. Il utilisa par conséquent une unité de mesure du produit indépendante de toute technique et langage de programmation, appelée le point de fonction (PF).

Le point de fonction est l'unité de mesure de la valeur fonctionnelle du produit livré à l'utilisateur indépendamment de toutes considérations techniques d'implémentation.

C'est de cette manière qu'Albrecht est original dans son calcul de productivité par rapport aux méthodes classiques décrites dans la première partie : en effet, celles-ci comptabilisent à postériori le nombre de LOC et dégagent l'effort nécessaire par une formule du type $E = a\ KLOC^b$. Ceci constitue la démarche inverse de celle suivie par Albrecht.

2. Le modèle d'application utilisé dans la méthode.

2.1. Introduction.

Pour rappel, le PF est l'unité de mesure de la valeur fonctionnelle du produit livré à l'utilisateur. Toute application à estimer sera analysée de manière fonctionnelle, c'est-à-dire que l'on décrira une application par un flux de données en entrée, un flux de données en sortie et des groupes de données logiques (GDL). Un GDL est un ensemble de données enregistrées dans une application et utiles aux traitements. De plus, ces données sont considérées au niveau logigue, sans considérations d'implémentation.

De tout ceci, il ressort qu'il y a un "intérieur" et un "extérieur", entre lesquels se trouve la frontière imaginaire de l'application. Seules les fonctions qui traversent cette frontière et les GDL sont considérés. Nous pouvons dire qu'il s'agit en quelque sorte de l'aspect visible de l'application ou encore, selon les termes d'Albrecht, de ses "manifestations extérieures". Le reste se trouve dans une black box.

Albrecht propose un modèle d'application dans lequel il dénombre cinq types de fonctions :

- les fonctions d'entrée externes (inputs)
- les fonctions de sortie externes (outputs)
- les fonctions de requête externes (inquiries)

- les fonctions d'interface externes (interfaces)

- les fonctions de groupes de données logiques internes (master files).

Cependant, pour une juste compréhension, le concept de "fonction" doit être interprété comme étant d'un intérêt fonctionnel pour l'utilisateur de l'application et non comme étant un traitement au sens informatique ou une fonction au sens mathématique du terme. On peut regretter l'ambiguïté qui plane autour de ce mot "fonction", mais nous pensons qu'Albrecht l'a choisi délibérément pour rappeler l'aspect fonctionnel de sa méthode.

Par conséquent, l'utilisation de la méthode se limite aux développement d'applications administratives. En effet, dans celles-ci, l'accent est placé sur des fonctions extérieures (applications interactives), tandis que les applications mathématiques placent plutôt l'accent sur des fonctions intérieures (peu d'interactivité).

Le modèle est illustré en figure 33

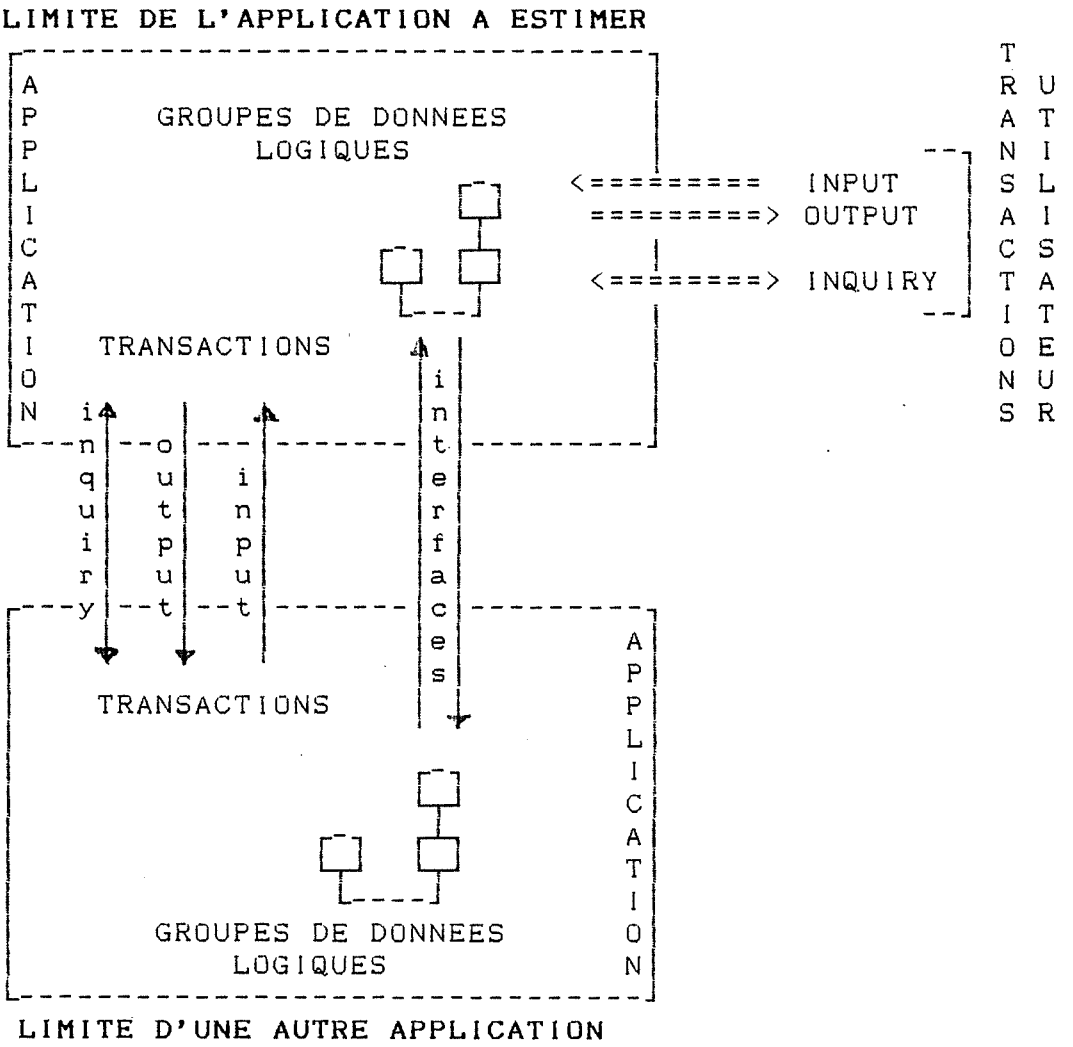


FIGURE 33 MODELE D'APPLICATION D'ALBRECHT

L'application communique avec deux types d'environnement :

- l'environnement "utilisateur"

- l'environnement "autre application".

Avec l'environnement "utilisateur", la communication s'effectue à l'intérieur de transactions composées d'inputs, d'outputs et d'inquiries. Par contre, avec l'environnement "autre application", la communication s'effectue non seulement par des transactions mais aussi par des interfaces.

La caractéristique d'une transaction est qu'elle décrit un événement et qu'elle déclenche un processus dans l'application mesurée. Il y a un côté actif dans la notion de transaction contrairement au côté passif d'un GDL qui décrit plutôt un état.

2.2. Définition d'une fonction d'entrée externe (input).

Une input fournit à l'application à mesurer de l'information en provenance de l'utilisateur ou d'une autre application.

Une input peut être utilisée pour ajouter, modifier ou consulter des données de l'application à mesurer (ceci constitue le côté actif d'une transaction).

2.3. Définition d'une fonction de sortie externe (output).

Une output fournit de l'information en provenance de l'application à mesurer vers l'utilisateur ou une autre application.

Cette information ne subira plus aucun autre traitement (sinon on parlerait d'interface).

2.4. Définition d'une fonction d'interface externe (interface).

Une interface est un GDL qui est créé à l'intérieur de l'application à mesurer et passé à d'autres applications ou qui est créé par d'autres applications et ensuite passé à l'application à mesurer.

2.5. Définition d'une fonction de groupe de données logique interne (master file).

Un master file est un GDL à la disposition de l'utilisateur qui est créé, utilisé et maintenu par l'application ou par l'utilisateur.

Par "maintenu", on entend l'ajout, la consultation ou la modification de données. Par "interne", on entend que ce GDL est propre à l'application à mesurer, c'est-à-dire non partagé avec d'autres applications.

2.6. Définition de la fonction de requête externe (inquiry).

Une inquiry est une combinaison d'input/output dans laquelle une input on-line déclenche directement une output on-line.

L'input ne déclenche qu'une action de consultation, en d'autres mots, aucun GDL n'est modifié.

2.7. La notion de complexité d'une fonction.

A l'aide des cinq fonctions que l'on vient de définir, il devrait être possible de décrire les "manifestations extérieures" de toute application.

Chacune des fonctions du modèle peut être classée dans un des trois niveaux de complexité suivants : simple, moyen ou difficile. Cette complexité est en fait un nombre de PF qui augmente quand la complexité augmente (cfr tableau 19).

FONCTIONS	SIMPLE	MOYEN	DIFFICILE
Master file	7	10	15
Input	3	4	6
Output	4	5	7
Inquiry	3	4	6
Interface	5	7	10

TABLEAU 19 LES VALEURS DES COMPLEXITES

2.8. Attribution du niveau de complexité.

On attribue une complexité à une input, output ou inquiry en fonction du nombre de types de données et du nombre de master files concernés.

Par "type de données", on entend un composant d'une input, d'une output ou d'une inquiry sémantiquement différent des autres composants. Par exemple, dans l'input suivante :

DUPONT JEAN
rue de la Charrue, 3
5.800 GEMBLOUX
081/615409

Il existe trois types de données sémantiquement différents : le nom, l'adresse et le numéro de téléphone.

Par contre, on attribue une complexité à un master file ou à une interface en fonction du nombre de types de records et du nombre total de champs dans ces records.

On peut remarquer que le niveau de complexité d'une fonction est dépendant du nombre d'objets manipulés (master files, interfaces, types de données, champs, types de records). Plus ce nombre d'objets est grand, plus le nombre de PF attribué est grand, et par conséquent, cette fonction a une valeur fonctionnelle plus élevée.

3. Les principes de fonctionnement de la méthode.

Le calcul proprement dit du nombre de points de fonction représentant la valeur fonctionnelle de l'application comporte deux étapes :

- le nombre de points de fonction brut
- le nombre de points de fonction net.

3.1. Le nombre de points de fonction brut.

Le nombre de points de fonction brut représente la valeur fonctionnelle de l'application d'un point de vue utilisateur, mais sans tenir compte de l'influence des contraintes de l'environnement sur les traitements de l'application (en quelque sorte, c'est calculer la valeur fonctionnelle du "quoi" sans tenir compte du "comment", par exemple, la valeur

fonctionnelle d'une application sans tenir compte du fait qu'elle sera répartie sur plusieurs sites).

Pour réaliser ce calcul brut, la méthode propose un poids à attribuer aux complexités (simple, moyenne et difficile) de chaque fonction (Input, Output,...). Ces poids ont été établis par essai et erreur durant plusieurs années d'expérimentation dans un environnement IBM. Ces poids sont des facteurs multiplicateurs. Par exemple, si on dénombre dans une application 5 Inputs simples, 3 Interfaces moyens, 2 Outputs difficiles et 1 Master File moyen, il suffit très simplement de les multiplier par le poids représentant leur complexité (cfr. figure 34 pour les valeurs de poids) et d'en faire la somme pour obtenir le nombre de points de fonction brut :

5 Inputs (simples)	*	3	= 15
3 Interfaces (moyens)	*	7	= 21
2 Outputs (difficiles)	*	7	= 14
1 Master File (moyen)	*	10	= 10
nombre de PF Brut			60

INPUTS	_ _ _	Simple	* 3	=	_ _ _
	_ _ _	Moyen	* 4	=	_ _ _
	_ _ _	Difficile	* 6	=	_ _ _
Totaux :	_ _ _				_ _ _
OUTPUTS	_ _ _	Simple	* 4	=	_ _ _
	_ _ _	Moyen	* 5	=	_ _ _
	_ _ _	Difficile	* 7	=	_ _ _
Totaux :	_ _ _				_ _ _
MASTER FILES	_ _ _	Simple	* 7	=	_ _ _
	_ _ _	Moyen	* 10	=	_ _ _
	_ _ _	Difficile	* 15	=	_ _ _
Totaux :	_ _ _				_ _ _
INTERFACES	_ _ _	Simple	* 5	=	_ _ _
	_ _ _	Moyen	* 7	=	_ _ _
	_ _ _	Difficile	* 10	=	_ _ _
Totaux :	_ _ _				_ _ _
INQUIRIES	_ _ _	Simple	* 3	=	_ _ _
	_ _ _	Moyen	* 4	=	_ _ _
	_ _ _	Difficile	* 6	=	_ _ _
Totaux :	_ _ _				_ _ _

NOMBRE DE POINTS DE FONCTION BRUT : _ _ _

FIGURE 34

3.2. Le nombre de points de fonction net.

Il s'agit ici de tenir compte des caractéristiques de l'environnement ainsi que des critères de performance et de qualité qui influencent la difficulté du développement de l'application. Ce sont en fait 14 caractéristiques dont l'influence est appréciée. En sommant les poids de chacune des 14 caractéristiques, on obtient le degré d'influence (DI) sur le développement de l'application. Le DI sert à calculer la valeur d'ajustement (VA) qui permettra d'obtenir le nombre de points de fonction net à partir du nombre de points de fonction brut :

La valeur d'ajustement (VA) : 0.65 + (0.01 * DI) = VA

Le nombre de PF Net : nombre de PF Brut * VA

Etant donné que les 14 caractéristiques ont des poids variant de 0 à 5, on peut donc obtenir pour DI une valeur variant de 0 à 70. Selon la formule de la valeur d'ajustement, on constate que VA peut prendre des valeurs allant de 0.65 à 1.35. Par conséquent, l'influence de l'environnement et des critères de performance/qualité peut corriger le nombre de points de fonction brut de -35 à +35 % ; on obtient ainsi le nombre de points de fonction net.

Nous allons maintenant expliquer comment dénombrer les cinq fonctions et comment déterminer leur niveau de complexité.

CHAPITRE 2.

DIRECTIVES POUR DENOMBRER LES TYPES DE FONCTIONS ET LEUR ATTRIBUER UN NIVEAU DE COMPLEXITE.

1. Les inputs.

Il faut compter comme input toute information en entrée de l'application à condition que :

- elle soit originaire de l'utilisateur ou d'une autre application. Des informations en entrée fournies par l'utilisateur uniquement pour des raisons techniques ne sont pas comptées comme input. Par exemple, un écran menu dans lequel l'utilisateur indique sa volonté d'utiliser telle option du programme n'est pas une input car cette information n'apporte rien du point de vue fonctionnel à l'application,

- elle possède la caractéristique d'unicité : une fonction input doit être considérée comme unique si elle a un autre format que les autres inputs ou si elle a le même format qu'une autre mais qu'elle requiert un autre traitement; un autre traitement, c'est soit différents master files modifiés, soit les mêmes master files mais modifiés de manière différente. Par même format, on entend le même nombre et les mêmes types de données,

- elle traverse la frontière imaginaire de l'application, c'est la caractéristique d'externalité,

- elle provoque l'ajout ou la modification de données dans un master file. Des exceptions sont pour autant possibles : il est possible de compter comme input une information en entrée n'occasionnant aucune modification dans un master file mais occasionnant une modification dans une interface. C'est la raison pour laquelle le tableau des complexités offre la possibilité de 0 master file.

1.1. Les niveaux de complexité.

A chaque input est attribué un des trois niveaux de complexité suivants :

- simple : peu de types de données sont introduits et peu ou pas de master files sont modifiés par ces inputs. De plus, la caractéristique de convivialité intervient peu dans le développement de l'input.

- moyen : l'input n'a ni une complexité simple, ni difficile.

- difficile : beaucoup de types de données sont introduits et beaucoup de master files sont modifiés. Il y a de plus un souci de convivialité.

Le tableau suivant aide à déterminer le niveau de complexité :

NBRE DE TD / NBRE DE MF	1 à 4	5 à 15	16 et +
0 à 1	S	S	M
2	S	M	D
3 et +	M	D	D

TD : Type de données
MF : Master file
S : Simple
M : Moyen
D : Difficile

TABLEAU 20

Exemple d'utilisation : si une input est composée de 7 types de données différents et qu'elle concerne deux master files, alors on attribue une complexité moyenne à cette input.

1.2. Correction en raison de considérations techniques.

Si on s'arrête au niveau de l'attribution des complexités, on pourrait négliger certaines considérations techniques qui influencent la réalisation. C'est pourquoi on introduit des facteurs supplémentaires qui, selon leur influence, feront passer la complexité de l'Input considérée à un niveau supérieur ou inférieur :

Exemples :

- positionnement automatique du curseur
- d'autres facteurs ergonomiques
- conversion de données
- performance de l'application.

Dans la version de base d'Albrecht, était déja prévue une correction de la complexité en fonction de ce genre de facteurs influençant le traitement de l'Input. Attention, ces facteurs introduisent des jugements subjectifs dans la méthode. C'est pourquoi il serait sans doute préférable de reporter ce genre

de considérations au niveau des 14 facteurs décrivant la complexité de l'application dans son entièreté.

1.3. Quelques inputs possibles.

- Documents introduits via le clavier
- Ecrans de données
- Cartes perforées
- Cartes magnétiques
- Crayons optiques
- Transactions en provenance d'autres applications
- Transactions sur bande, sur disquette
- Les touches-fonction

1.4. Conseils pour le comptage.

- Données introduites par l'écran (= 1 Inp)
- Les transactions de données à partir d'autres applications (= 1 Inp)
- Résultat d'une requête préalable servant à une modification (= 1 Inp)
- des touches fonctions(ou crayon optique) faisant double emploi avec un écran déja compté comme Imput (= 0 Inp)
- Deux écrans de données avec le même format et le même traitement (=1 Inp)
- Deux écrans de données avec le même format et deux traitements différents (= 2 Inp)
- Ecran menu (= 0 Inp)
- Ecran menu avec possibilité de sauvetage (= 1 Inp)
- Un écran qui sert aussi bien d'entrée que de sortie (= 1 Inp et 1 Out)
- Un formulaire comme entrée; par exemple, un formulaire de mutation de personnel pourra être utilisé aussi bien pour un engagement que pour une promotion ou un transfert. A partir d'un seul

formulaire, différentes Inputs peuvent être comptées séparément (données différentes, traitements différents) (= au moins 1 Inp).

2. Les outputs.

Il faut compter comme output toute information en sortie de l'application et destinée à l'utilisateur ou à une autre application, à condition qu'elle respecte les caractéristiques d'unicité et d'externalité.

Nous comptons les outputs à l'écran qui produisent des messages d'erreur ou des messages à l'opérateur, sauf si le message est un simple accusé de réception d'une transaction d'entrée ou sauf s'il s'agit d'un message d'erreur qui ne requiert pas de traitement autre qu'une édition.

Nous ne compterons pas les écrans outputs qui sont nécessaires au système pour une implémentation spécifique; par exemple, des écrans qui sont produits uniquement dans le but d'obtenir d'autres écrans, et qui ne produisent pas d'information pour l'utilisateur.

Un GDL en sortie de l'application n'est pas compté comme une output mais bien comme une interface.

2.1. Les niveaux de complexité.

Le niveau de complexité d'une output est déterminé selon le même principe que celui des inputs. Seul le tableau des complexités est modifié.

NBRE DE TD / NBRE DE MF	1 à 5	6 à 19	20 et +
0 à 1	S	S	M
2 à 3	S	M	D
4 et +	M	D	D

TD : Type de données
MF : Master file
S : Simple
M : Moyen
D : Difficile

TABLEAU 21

En particulier, on peut donner les définitions complémentaires suivantes :

- simple : il n'y a qu'une ou deux colonnes dans l'output et il s'agit de simples transformations de données,

- moyen : il y a plus de colonnes, avec des sous-totaux et un plus grand nombre de transformations de données,

- difficile : il y a de nombreuses transformations de données, de nombreuses et complexes utilisations de master files. De plus, des considérations de performance sont envisagées.

2.2. Corrections en raison de considérations techniques.

On peut également procéder à des corrections pour des raisons techniques telles que :

- des facteurs ergonomiques pour l'affichage,

- des transformations de données,

- la performance de l'application.

2.3 Quelques outputs possibles.

- rapports à l'écran
- rapports batch
- transactions sur bande, sur carte perforée ou sur disquette
- simple information à l'écran
- transaction vers autre application
- factures
- chèques
- listings

2.4. Conseils pour le comptage

- une sortie de données sur l'écran (= 1 Out)
- les transactions de données en faveur d'autres applications (= 1 Out)
- un écran de plusieurs messages d'erreur ou de confirmation lié à une seule Input (= 1 Out)
- un seul message d'erreur ou de confirmation (= 0 Out)
- un rapport(= 1 Out). Cependant, celui-ci peut contenir plusieurs Outputs: c'est par exemple le cas si un rapport avec une information détaillée est suivi par un résumé de plusieurs pages. Ce dernier contient souvent des données complémentaires et de plus, requiert un autre traitement. Dans une telle situation, on parle de deux ou plusieurs Outputs.
- listing (= 1 Out)
- un menu en sortie (= 0 Out)

3. Les master files.

Nous compterons comme master file chaque GDL qui est créé, utilisé ou maintenu par l'application ou l'utilisateur.

Nous insisterons sur le fait qu'il faut compter des fichiers logiques du niveau fonctionnel et non pas les fichiers physiques.

3.1. Les niveaux de complexité.

A chaque mater file est attribué un des trois niveaux de complexité suivants :

- simple : il y a peu de types de records et peu de champs. De plus, des considérations de performance et de récupératon sur incident ne sont pas envisagées,
- moyen : c'est entre simple et difficile,
- difficile : il y a beaucoup de types de records et beaucoup de champs. De plus, des exigences de performance et de récupération sur incidents sont posées.

NBRE DE CHAMPS / NBRE DE TR	1 à 19	20 à 50	51 et +
1	S	S	M
2 à 5	S	M	D
6 et +	M	D	D

TR : Type de records
S : Simple
M : Moyen
D : Difficile

TABLEAU 22

Le concept de type de records fait partie de vieilles méthodes et techniques de développement de systèmes. Par conséquent, il ne devrait plus être utilisé. Le nombre de types de records sera quasi toujours égal à 1.

3.2. Corrections en raison de considérations techniques.

On peut également procéder à des corrections pour des raisons techniques telles que :

- les performances de l'application,
- des critères de recherche,
- des exigences de récupération et de back up.

3.3. Les master files possibles.

- Les bases de données ou sous-schémas de la base de données
- Tables-utilisateurs (vues).

3.4. Conseils pour le comptage.

- une entité logique de données d'un point de vue utilisateur (= 1 MF)
- groupe de données logique qui est créé et maintenu par l'application (= 1 MF)
- fichiers intermédiaires (non permanents) fichiers triés.(= 0 MF)

4. Les interfaces.

Il faut compter comme interface les GDL qui circulent entre des applications, ou qui sont utilisés en commun par des applications. Autrement dit, ce sont des GDL qui sont créés par l'application à estimer et passés à d'autres applications, ou créés par d'autres applications et passés à l'application à estimer. Ils possèdent également la caractéristique d'externalité.

4.1. Les niveaux de complexité.

A chaque interface est attribué un des trois niveaux de complexité suivant les mêmes principes que ceux adoptés pour la complexité des master files.

4.2. Corrections en raison de considérations techniques.

On peut également procéder à des corrections pour les raisons techniques suivantes :

- les performances de l'application,
- les critères de recherche,
- des exigences de récupération et de back up,
- le nombre d'applications avec lesquelles le GDL est interface.

4.3. Les interfaces possibles.

1 : un GDL accessible par une autre application

2 : un GDL appartenant à une autre application et accessible par l'application considérée

3 : une base de données commune

La première et la dernière possibilité sont comptées comme MF parce que l'application les gère et les modifie; de plus, ils sont interfaces car ils peuvent être utilisés par d'autres applications.

Possibilités	L'application	L'autre application
1	MF et interface	seulement interface
2	seulement interface	interface et MF
3	MF et interface	MF et interface

4.4. Conseils pour le comptage.

- un groupe de données logique géré par une autre application (= 1 Int)
- un groupe de données logique géré par l'application et utilisé par d'autres applications (= 1 Int)
- une base de données d'une autre application dont il est fait usage dans l'application (= 1 Int)
- la base de données de l'application, utilisée par d'autres applications (= 1 Int)

Etant donné qu'une base de données peut inclure plusieurs groupes de données logiques, on compte une interface par groupe de données logique utilisé.

5. Les inquiries

Il faut compter comme inquiry chaque combinaison d'input/output où une input cause la création directe d'une output, pour autant qu'elle respecte les caractéristiques d'unicité et d'externalité.

Il faut compter aussi bien les inquiries déclenchées par l'utilisateur que celles qui sont déclenchées par d'autres applications.

On distinguera les inputs des inquiries par le fait que ces dernières provoquent exclusivement des consultations, par conséquent, elles n'effectuent aucune modification sur les GDL. On veillera à ne pas confondre une "query facility" avec une inquiry. Une inquiry est une requête directe concernant des données spécifiques, en général, n'utilisant qu'une seule clé d'accès. Par contre une "query facility", travaille sur plusieurs clés d'accès et dès lors nécessite plusieurs inputs, outputs et inquiries.

5.1. Les niveaux de complexité.

A chaque inquiry sera attribué un niveau de complexité selon les règles suivantes :

- attribuer une complexité à la partie input de l'inquiry en utilisant le tableau des complexités des inputs et une complexité à la partie output de l'inquiry en utilisant le tableau des complexités des outputs

- la complexité d'une inquiry est alors déterminée par la plus forte complexité attribuée à l'une des deux parties de l'inquiry.

5.2. Les inquiries possibles.

- les écrans et messages d'aide (help)

- les écrans de menu

5.3. Conseils pour le comptage.

- une entrée on-line et une sortie on-line sans modification de données dans les MF (= 1 Inq)
- une interrogation suivie d'une entrée qui modifie un MF (= 1 Inq + 1 Inp)
- apparition d'écran d'aide (= 1 Inq)
- apparition d'écran-menu (= 1 Inq)
- "query-facility" (= plusieurs Inq., plusieurs Inp.et plusieurs Out à déterminer)

Par les inquiries, nous terminons ainsi la description des différentes fonctions qui entrent en compte pour l'estimation de l'application. Nous avons également cité plusieurs directives et conseils pratiques pour l'identification des différentes fonctions. Maintenant, nous décrivons les caractéristiques globales de l'application qui permettent d'ajuster l'estimation.

6. Les 14 caractéristiques ou facteurs d'ajustement.

6.1. Télécommunication.

Les données qui sont utilisées dans l'application sont envoyées ou reçues via des moyens de télécommunication.

N.B.: les terminaux qui sont reliés localement à l'ordinateur, ne doivent pas être repris.

6.2. Données distribuées ou traitements distribués.

Il s'agit de systèmes répartis.

6.3. Performances.

Le système de développement, l'implémentation et la maintenance de l'application sont influencés par des objectifs de temps de réponse ou de "throughput" (déterminé par l'utilisateur ou approuvé par celui-ci).

6.4. Charge.

L'application doit "tourner" sur un ordinateur qui est déjà lourdement chargé (charge pendant l'exécution), ce qui entraîne des considérations spécifiques pour le développement du projet.

6.5. Volume de transactions.

Le volume de transactions est élevé et influence le développement du système, son implémentation et sa maintenance.

6.6. Data-entry.

L'application est du type "on-line data entry" et offre des fonctions de contrôle.

6.7. Aspect conversationnel de l'application.

Des fonctions on-line sont créées dans l'application pour atteindre un haut niveau d'efficacité du côté utilisateur.

6.8. Mise-à-jour de données en "on-line".

La MAJ on-line des MF et des bases de données de l'application.

6.9. Traitements internes complexes.

Cela peut être le cas si des traitements logiques ou mathématiques sont réalisés. On peut également penser aux soucis de sécurité qui influencent le traitement.

6.10. Réutilisabilité.

L'application et sa programmation sont réalisées, développées et maintenues de telle manière qu'elles peuvent être réutilisées dans d'autres applications.

6.11. La mise en service.

L'application comprend une phase importante de travaux de conversion et/ou de démarrage.

6.12. Commodités de service.

Les procédures de démarrage, de sauvetage et de récupération sont créées et testées dans le développement du système.

L'application minimise les activités manuelles durant son exécution (degré d'automatisation élevé).

6.13. Localisation-organisation.

L'application est développée et maintenue de manière telle qu'elle soit utilisée en des sites différents au profit de plusieurs organisations.

6.14. Flexibilité.

L'application est réalisée de manière telle qu'elle permet à l'utilisateur d'apporter des modifications aisément.

Exemples :

- l'utilisateur peut facilement modifier les données et paramètres de l'application via des tableaux.

- l'utilisateur bénéficie de beaucoup de possibilités de requêtes.

CHAPITRE 3.

CONSEILS PRATIQUES.

1. Introduction.

Dans cette partie, nous citerons un certain nombre de conseils utiles pour le comptage des fonctions, ceci afin d'éclairer les parties ambiguës laissées par la théorie.

Lorsque sera employée la méthode des points de fonction au sein d'une entreprise, des questions surgiront sûrement quant à la manière de l'appliquer. Il faut espérer que les directives reprises dans cette partie pourront aider à résoudre un certain nombre de problèmes.

De manière générale, il est recommandé à l'utilisateur d'avoir en tête la philosophie de cette méthode afin d'obtenir une estimation la plus consistante possible.

On ne peut pas espérer que deux personnes obtiennent le même résultat mais qu'une personne soit cohérente pour estimer différentes applications.

2. Directives générales.

Un danger de la méthode est de comptabiliser plusieurs fois les mêmes choses; nous conseillons par conséquent de se poser régulièrement la question suivante : cette fonction n'est-elle pas déja comptée autre part?

La réalisation technique ou l'implémentation n'est pas prise en considération dans la méthode. Celle-ci veut se situer à un point de vue logique et non physique.

Le bon sens dans l'application des directives reste nécessaire.

3. Inputs.

La complexité d'une input est déterminée par le nombre de types de données et le nombre de MF employés. Nous présenterons ici quelques directives :

1. La connaissance du nombre de types de données et du nombre de MF est dépendante de la phase du développement dans laquelle le projet se trouve. Si ces données ne sont pas connues, la complexité moyenne sera attribuée à chaque input identifiée.
   N.B : Oublier de compter une fonction est plus

grave pour une estimation correcte que de sous-estimer sa complexité.

2. En ce qui concerne le calcul de types de données, voici les directives proposées :

   - tous les types de données employés pour valider ou modifier les MF doivent être pris en compte.

   - ainsi que tous les types de données qui sont employés pour produire des messages d'erreur ou accusés de réception.

3. Le nombre de MF est déterminé par le nombre de groupes de données logiques utilisés pour une validation et le nombre de groupes de données logiques utiles à l'exécution d'une transaction.

DIRECTIVES COMPLEMENTAIRES :

- il faut qu'il y ait au moins une input liée à chaque MF pour effectuer une lecture ou une modification. Vérifier cette règle permet au développeur de détecter l'oubli d'inputs de la part de l'utilisateur. A quoi servirait un MF s'il n'existe pas au moins une input pour y accéder ?

- pour un même écran utilisé pour plusieurs traitements (entrées de données, modification, suppression), il faut compter une input par traitement et relever séparément le nombre de types de données et le nombre de MF employés par chaque traitement.

- la suppression de données en deux phases, c'est-à-dire l'ordre de suppression suivi de la confirmation sera compté comme une seule input.

- les différents champs d'un écran ne constituent pas différentes inputs, mais sont les types de données qui interviennent dans la détermination de la complexité de l'écran-input.

4. Outputs.

En ce qui concerne les types de données, voici les directives proposées :

- tous les types de données qui se présentent dans l'output (pas les valeurs des données).

- les données du traitement, ainsi que les résultats des multiplications, sous-totaux, etc...

- les données de sélection qui, afin de produire l'output, doivent être introduites (pour autant qu'il ne s'agisse pas d'une inquiry).

- par contre, les descriptions de colonnes, d'entêtes et littéraux ne sont pas considérés comme des types d'entrées.

DIRECTIVES COMPLEMENTAIRES :

- si un rapport fait l'objet de plusieurs sorties (par exemple, plusieurs classifications, niveaux de détails différents, résumés d'une ouput), chacune de ces sorties doit être comptée séparément comme output.

- une sortie, qui a la même structure mais qui peut être triée de plusieurs manières, doit être comptée comme une seule output à moins que le critère de tri requière un traitement supplémentaire.

- des groupes de données logiques destinés à d'autres applications sont des interfaces et non des outputs.

- ne pas confondre la partie output d'une inquiry avec une output.

- on ne compte pas les messages d'erreur ou accusés de réception liés aux inputs, sauf s'ils font l'objet d'une présentation sur un écran particulier. De plus, il sera tenu compte de ce genre de facilités dans le facteur de correction numéro 7 : aspect conversationnel de l'application.

5. Les groupes de données logiques internes (MF)

Un groupe de données logique quelconque, ni généré, ni modifié par l'application à mesurer est soit une interface, s'il provient d'une autre application, soit un groupe de données logique sur lequel, à tort, aucun traitement n'a été défini. Dans ce cas douteux, il est recommandé de consulter l'utilisateur.

DIRECTIVES :

- pour chaque MF, on doit s'attendre à avoir au moins une Input, une Output ou une Inquiry.

- un groupe de données logique créé à la demande de l'utilisateur pour assurer la récupération, par exemple, ne doit pas être compté. On en tiendra

compte dans les facteurs de correction, à savoir "MAJ on-line" et "commodités de service".

- si un groupe de données logique est trié suivant plusieurs critères, celui-ci sera cependant compté qu'une seule fois.

6. Les interfaces.

La complexité d'une interface est déterminée de la façon suivante : il faut seulement compter les champs et les types de records que l'application utilise et non pas nécessairement l'entièreté.

DIRECTIVES :

- les groupes de données logiques qui sont à la fois modifiés par l'application et aussi partagés avec d'autres applications seront considérés comme MF et interfaces.

- une interface doit être un MF dans une autre application.

- on ne doit pas considérer un groupe de données logique comme interface s'il est utilisé entre deux sous-systèmes de l'application, car un groupe de données logique est considéré comme interface seulement s'il dépasse la frontière imaginaire de l'application.

7. Les inquiries.

La complexité des inquiries est déterminée par leur partie input et par leur partie output (voir les directives des inputs et outputs)

8. Les facteurs de correction.

A la première étape (la détermination du poids des facteurs), l'échelle suivante est appliquée :

0 : pas d'influence

1 : faible influence

2 : influence modérée

3 : influence normale

4 : influence significative

5 : forte influence.

Les valeurs ci-dessus sont des directives pour mesurer la valeur de l'influence.

Ces 14 facteurs ont été choisis parce qu'ils ont une influence caractéristique sur le développement, sur la documentation, sur la maintenance de l'application. Ces facteurs devraient être décrits directement ou indirectement dans la documentation ou dans les spécifications fonctionnelles.

Voici les directives proposées par la firme Audit Informatica [Audit **] :

8.1. Télécommunication.

Les données qui sont utilisées dans l'application sont envoyées ou reçues via des moyens de télécommunication.

N.B : les terminaux qui sont reliés localement à l'ordinateur, ne doivent pas être repris.

0 : si l'application consiste exclusivement en un travail batch

1-2 : on parle d'imprimantes placées sur un site distant et/ou d'entrées de données à partir d'un site distant

3-5 : on parle de "téléprocessing" interactif (TP)

ex : 3 = si le TP est "front-end" vers un processus batch

5 = si l'application consiste principalement en TP.

8.2. Données distribuées ou traitements distribués.

0 : l'application ne soutient pas la transmission de données ou de traitements entre composants d'un système réparti

1 : l'application prépare des données pour un traitement terminal sur un autre composant du système réparti

2-4 : les données sont préparées pour le transfert, envoyées et ensuite, traitées sur un autre composant du système réparti

5 : les fonctions de traitement sont dynamiquement exécutées sur le composant du système le plus apte à le faire.

8.3. Performances.

Le système de développement, l'implémentation et la maintenance de l'application sont influencés par des objectifs de temps de réponse ou de "throughput" (déterminé par l'utilisateur ou approuvé par celui-ci).

0-3 : L'analyse de performance et les considérations de développement sont standards. Aucune exigence de performance n'est posée par l'utilisateur

4 : Les activités d'analyse de performance sont enregistrées dans la phase de développement afin de pouvoir satisfaire les objectifs de performance spécifiés par l'utilisateur

5 : En complément, des moyens d'aide à l'analyse de performance sont appliqués pendant les phases d'ébauche, de développement et d'installation du projet afin de satisfaire les exigences de performance spécifiées par l'utilisateur.

8.4. Charge.

L'application doit "tourner" sur un ordinateur qui est déjà lourdement chargé (charge pendant l'exécution), ce qui entraîne des considérations spécifiques pour le développement du projet.

0-3 : L'application "tourne" sur une machine de production standard. Il n'y a pas de limites opérationnelles fixées

4 : Certaines limites opérationnelles posent des exigences spéciales à l'application

5 : De plus, ici, des limites spécifiques sont posées à un autre composant du système (distribué). Un composant du système réparti, sur un site éloigné, doit également faire face à des surcharges.

8.5. Volume de transactions.

Le volume de transactions est élevé et influence le développement du système, son implémentation et sa maintenance.

0-3 : Le degré de transaction est tel que les considérations d'analyse de performance sont standards

4 : Les tâches d'analyse de performance font partie de la phase d'ébauche du projet, afin de pouvoir satisfaire à de grands volumes de transactions comme ils ont été spécifiés par l'utilisateur

5 : En plus, ici, sont utilisés des moyens d'aide à l'analyse de performance dans les phases d'ébauche, de développement et d'installation du projet

8.6. Data-entry.

L'application est du type "on-line data entry" et offre des fonctions de contrôle.

0-2 : 0 à 15 % des transactions sont des entrées de données interactives

3-4 : 15 à 30 % des transactions sont des entrées interactives

5 : 30 à 100 % des transactions sont des entrées interactives

8.7. Aspect conversationnel de l'application.

Des fonctions "on-line" sont créées dans l'application pour atteindre un haut niveau d'efficacité du côté utilisateur.

0-3 : Il n'existe aucune exigence de la part de l'utilisateur (au niveau de la convivialité et de l'efficacité)

4 : Dans la phase d'ébauche du projet, des tâches ont été enregistrées pour tenir compte des facteurs ergonomiques afin de tenir compte des exigences de l'utilisateur

5 : En plus, ici, des moyens spéciaux d'aide (prototypage) sont employés pour fournir les critères de convivialité et d'efficacité

8.8. Mise-à-jour des données en "on-line".

La MAJ on-line des MF et des bases de données de l'application.

0 : Pas de mise-à-jour "on-line"

1-2 : Les MF sont manipulés "on-line". Leur volume est petit, et la récupération sur incident est simple

3 : Manipulation "on-line" de MF importants

4 : Manipulation "on-line" de MF importants, mais en plus, la sécurité contre la perte de données est une caractéristique essentielle

5 : Idem que 4, mais en plus interviennent les aspects de coût dans les considérations de récupération en rapport avec le volume des MF.

8.9. Traitements internes complexes.

Cela peut être le cas si des traitements logiques ou mathématiques sont réalisés. On peut également penser aux soucis de sécurité qui influencent le traitement.

0 : aucune des deux caractéristiques citées ci- dessus n'est d'application

1-3 : Une des caractéristiques est d'application

4-5 : Les deux caractéristiques sont d'application

8.10. Réutilisabilité.

L'application et sa programmation sont réalisées, développées et maintenues de telle manière qu'elles peuvent

être réutilisées dans d'autres applications.

0-1 : Une application locale qui satisfait aux besoins d'une organisation utilisatrice

2-3 : L'application utilise des modules de programmes communs dans lesquels il est tenu compte des besoins de plusieurs utilisateurs et elle crée d'autres modules tenant compte des besoins d'autrui

4-5 : Idem que 2-3, mais en plus, l'application a été créée et documentée de telle manière que sa réutilisabilité soit simple.

8.11. La mise en service.

L'application comprend une phase importante de travaux de conversion et/ou de démarrage.

0-1 : Il n'y a pas de considérations spéciales sur la conversion et l'installation spécifiées par l'utilisateur

2-3 : Des exigences de conversion et d'installation sont spécifiées par l'utilisateur et des directives de conversion et d'installation sont fournies et testées

4-5 : Idem que 2-3, mais en plus, des moyens d'aide à la conversion et à l'installation sont disponibles et testés

8.12. Commodités de service.

Les procédures de démarrage, de sauvetage et de récupération sont créées et testées dans le développement du système.

L'application minimise les activités manuelles durant son exécution (degré d'automatisation élevé).

0 : Il n'y a pas d'aspects opérationnels spécifiés par l'utilisateur

1-2 : Des facilités effectives de démarrage, de sauvetage et de récupération sont fournies et testées

3-4 : Idem que 1-2, de plus, l'application minimise la nécessité d'activités manuelles (pose de bandes, manipulation de papier)

5 : L'application a été conçue comme une application "unattended operation"

8.13. Localisation-organisation.

L'application est développée et maintenue de manière telle qu'elle soit utilisée en des sites différents au profit de plusieurs organisations.

0 : L'utilisateur n'a pas exigé une utilisation sur plus d'un site

1-3 : Les besoins d'utilisation sur plusieurs sites sont considérés dans l'ébauche du projet

4-5 : Une documentation et un plan de maintenance sont fournis et testés afin de pouvoir maintenir l'application sur plusieurs sites

8.14. Flexibilité.

L'application est réalisée de manière telle qu'elle permet à l'utilisateur d'apporter des modifications aisément.

Exemples :

- l'utilisateur peut facilement modifier les données et paramètres de l'application via des tableaux.
- l'utilisateur bénéficie de beaucoup de possibilités de requêtes.

0 : Il n'y a pas d'exigences spéciales de l'utilisateur pour que l'application puisse supporter de simples modifications

1-3 : on prévoit des possibilités de requêtes flexibles

4-5 : Idem 1-3, de plus, possibilités de modifications de valeurs "on-line"

CHAPITRE 4.

JUGEMENT DE LA METHODE DES POINTS DE FONCTION.

En utilisant les critères définis dans la conclusion de la première partie, nous allons juger les caractéristiques de la méthode des points de fonction.

1. La définition.

Dans la description de la méthode, Albrecht définit quelles sont les heures réellement estimées. Il s'agit des heures consacrées au développement du logiciel par l'équipe de développement, à l'exclusion des heures consacrées à la définition du projet, à l'achat du logiciel et du hardware, à la maintenance et à la formation de l'utilisateur après la mise en production.

Il faut remarquer que la méthode ne constitue pas un modèle d'estimation des coûts. L'estimation totale des heures consacrées au projet ne donne pas une idée de la manière dont il faut répartir les heures entre les différentes équipes de développeurs dans les différentes phases du cycle de vie.

2. La fiabilité.

Il est encore difficile de se prononcer sur la fiabilité de la méthode étant donné le manque de publications à ce sujet. On peut dire simplement que dans certains environnements, des expériences ont été réalisées; celles-ci montraient que la méthode fournissait des résultats fiables [Rudolph 83] [Kemerer 87].

3. L'objectivité.

C'est sans contestation le point le plus faible de la méthode. La pondération des cinq fonctions selon les trois niveaux de complexité comprend une part de subjectivité. Même avec l'aide des tableaux de complexité, les pondérations des fonctions effectuées par différentes personnes divergent encore.

Il faut également relever les 14 facteurs de correction qui, par leur définition, prêtent à beaucoup de subjectivité.

4. La compréhensibilité.

La description de la méthode, telle qu'elle a été donnée par Albrecht dans son rapport original, était très succincte. En effet, celle-ci avait été décrite comme illustration pour montrer qu'il était possible de mesurer l'évolution de la productivité. Par conséquent, les concepts employés étaient définis de manière peu précise, et menaient à des problèmes d'interprétation.

Rudolph, cependant, a apporté des éclaircissements à cette méthode mais sans pour autant définir les concepts de manière plus précise. Il s'est contenté de la réexpliquer en donnant de nombreux exemples et de nombreuses directives. Il a d'ailleurs intitulé son rapport "cookbook". C'est en fait une réelle recette de cuisine.

C'est pour cette raison que nous avons essayé de dégager le modèle d'application sur lequel repose la méthode et de redéfinir les concepts utilisés pour réduire cette part de subjectivité. Par exemple, il nous a fallu un certain temps pour comprendre la signification qu'a le mot "fonction" dans la méthode. De plus, pourquoi avoir attribuer 3 PF au lieu de 1 PF à une input de complexité simple ?

Malgré ce plus grand degré de précision dans la définition des concepts, quand il s'agit d'appliquer concrètement la méthode, il est parfois bien difficile de classifier le flux d'information dans une des cinq catégories.

En fait ce problème de subjectivité n'est qu'une conséquence de ce manque de compréhensibilité.

Il est cependant facile de comprendre le mécanisme du calcul des PF car il se limite à de simples additions et multiplications.

Lorsque l'on utilise la méthode, il arrive un moment où on se pose la question de savoir ce que représente un PF. Il est bien difficile de répondre à cette question.

5. Les détails.

Appliquer la méthode à des sous-systèmes d'une application ne pose pas de problèmes.

6. La stabilité.

Le peu de résultats que l'on possède ne montre aucune discontinuité.

7. Le domaine d'application.

La méthode est orientée vers des applications administratives et n'est pas du tout applicable à des systèmes mathématiques.

8. La facilité d'utilisation.

Elle est bien sûr compromise étant donné les problèmes de compréhensibilité. Suivant la documentation consultée, le niveau de détails dans les explications des concepts de base est différent. Comme nous l'avons déjà dit, la méthode n'a pas été définie clairement par son auteur; on peut donc difficilement se référer à sa façon de faire.

Même si tout était clair et bien défini, l'effort à produire pour fournir des informations nécessaires en entrée de la méthode reste encore très grand.

9. L'aspect prévisionnel.

La méthode a été conçue pour fonctionner à postériori. Cependant, elle peut être utilisée plus tôt mais elle doit pour cela disposer d'une description logique des traitements et des données. L'endroit idéal serait la fin de l'analyse conceptuelle.

10. La pertinence et la complétude du choix des facteurs.

Se prononcer sur la pertinence et la complétude du choix des facteurs semble difficile. En effet, idéalement, une méthode d'estimation devrait englober l'ensemble des facteurs décrits en première partie de ce travail; cette liste, de plus, n'était pas exhaustive.

On peut simplement prétendre que les 14 facteurs retenus par Albrecht étaient ceux qui provoquaient dans son environnement le plus grand écart de productivité; c'est pourquoi, ils étaient les plus pertinents pour lui. Malgré ceci, la méthode semble être la plus transposable dans d'autres environnements.

11. La rapidité versus la qualité d'estimation.

Selon de Kater, il faut entre un et trois jours pour réaliser une estimation par la méthode des PF en fonction de la taille du projet; quant à sa qualité, selon l'étude de Kémerer, elle paraît être grande par rapport aux autres méthodes [de Kater 87] [Kemerer 87]. Dans la littérature, on observe un certain consensus au sujet de la précision des résultats.

Certains diront qu'un à trois jours de travail pour l'estimation, c'est trop long; d'autres diront que c'est raisonnable. Tout ceci dépend en fait de l'intérêt que l'on porte aux méthodes d'estimation.

12. La possibilité d'améliorer l'estimation avec l'expérience.

La méthode ne prévoit rien. Cependant, si l'estimation est toujours réalisée par la même personne, il est certain que cette dernière acquerra une meilleure connaissance de la méthode et des concepts, et de là, on peut supposer que les estimations s'amélioreront avec l'expérience.

13. L'indépendance vis à vis de son environnement d'expérimentation.

Selon l'étude de Kemerer, c'est la méthode qui donne les meilleures estimations dans des environnements fort différents [Kemerer 87]. On peut expliquer ceci par la volonté d'estimer un projet par une approche fonctionnelle et non physique.

Cependant, le choix des 14 facteurs de correction a certainement été guidé par l'environnement dans lequel Albrecht a mené son étude.

Partie 3

Adaptation et implémentation de la méthode des points de fonction.

CHAPITRE 1.

INTRODUCTION.

1. L'objectif du stage.

La C.G.E.R. nous a demandé d'étudier l'applicabilité de la méthode des PF. Celle-ci devrait permettre d'estimer le poids à attribuer à la charge de développement d'un nouveau projet. Ce poids pourrait être utilisé pour comparer les solutions alternatives d'un projet à développer ou différents projets entre eux.

De plus, cette estimation devait être réalisée à priori, c'est-à-dire dans les toutes premières phases du cycle de vie.

Et enfin, la C.G.E.R. nous a demandé également s'il était possible d'implémenter une telle méthode sur un micro-ordinateur.

2. Démarche suivie.

Etant donné que la méthode, telle qu'elle avait été proposée par Albrecht, fonctionnait à postériori, nous avons dû y apporter quelques modifications pour la rendre utilisable à priori.

Afin d'illustrer notre implémentation de la méthode des PF et de la tester, nous avons choisi l'application "Travellers Chèques" (TC) développée à la C.G.E.R.

Cependant, il est nécessaire de préciser la démarche que nous avons suivie pour l'étude de cette application. En effet, cette application est déjà mise en production. Mais, la méthodologie proposée par le service "Méthodologie et Administration de l'Information" (MAI) n'ayant pas été respectée scrupuleusement par les développeurs, nous n'avons pu que constater l'absence des modèles de l'analyse conceptuelle. C'est ainsi que nous avons été obligés d'élaborer, avec l'aide de messieurs Spiltoir et Graas du MAI, le modèle conceptuel des données à partir des spécifications des écrans et du contenu des fichiers. Nous reconnaissons, bien entendu, la précarité d'une telle démarche, mais cela constituait pour nous le seul moyen pratique de tester notre implémentation de la méthode des PF. De toute manière, un responsable du développement de l'application TC, monsieur van Renterghem, a reconnu la validité de notre modèle conceptuel.

3. Description générale de la vente de TC.

Nous avons travaillé à partir du rapport "Travellers Chèques 1987/133 197/INS. 54/A0".

"Le traveller chèque ou chèque de voyage est un titre nominatif dont la valeur nominale est libellée en monnaie étrangère; en voici les principales caractéristiques :

- négociable auprès de banques ou de commerçants
- possibilité de remboursement en cas de perte ou de vol pour autant que certaines conditions soient respectées
- aucune date limite de validité (sauf pour les lires italiennes)
- cours du TC spécifique et généralement plus avantageux que le cours du billet.

Le principe de fonctionnement.

Le client signe une première fois le TC lors de l'achat. Au moment de la présentation à l'encaissement, il contresignera ce chèque. La concordance entre les deux signatures est la garantie pour l'acheteur d'obtenir le paiement".

La vente.

Il existe quatre acteurs dans le scénario vente de TC :

- l'émetteur du TC
- le siège central
- l'agence
- le client acheteur du TC.

Le siège central commande via un bordereau de commande à un émetteur une quantité déterminée de TC, de devises et de coupures. L'émetteur livrera au siège central la quantité demandée et accompagnée d'un bordereau de livraison.

Le même principe de fonctionnement sera suivi entre l'agence et le siège central. Et enfin, l'agence vendra les TC à ses clients, ce qui donnera naissance à un bordereau de vente renvoyé vers le siège central.

Par l'intermédiaire des numéros de TC, des bordereaux de livraison, d'accompagnement et de vente, il est possible de

connaître exactement qui possède un TC déterminé, et d'en faire ainsi le suivi.

Avertissement : nous sommes bien conscients que cette description est fort générale, mais étant donné que l'application des TC nous sert uniquement d'illustration à notre travail, nous avons voulu donner uniquement les principes de fonctionnement afin que le lecteur puisse comprendre les modèles conceptuels de l'application.

CHAPITRE 2.

L'IMPLEMENTATION

DE LA METHODE DES PF.

1. Motivations.

Le principe fondamental qui a guidé notre implémentation de la méthode des PF est la facilité d'utilisation. En effet, nous avons remarqué, lors de nos entretiens avec plusieurs responsables de projet, que les méthodes d'estimation n'étaient guère appréciées en général, et que si certains voulaient bien faire preuve de bonne volonté, c'était dans certaines limites de temps et d'effort. Nous pouvons assez bien synthétiser le sentiment général sur la question par la phrase suivante : "pourquoi voudriez-vous que je consacre du temps et de l'effort à des méthodes d'estimation qui ne donnent pas de meilleures résultats que ceux obtenus grâce à l'intuition et l'expérience ?"

Etant donné la difficulté de réaliser des estimations correctes, et si on se borne à un aspect prévisionnel des méthodes d'estimation, il est certain qu'on ne peut pas en vouloir aux développeurs de réagir de la sorte. Selon nous, il faut voir les méthodes d'estimation non seulement comme un outil de prévision mais aussi comme un moyen d'obtenir une meilleure connaissance de l'application, et ce dès le début du développement.

C'est pourquoi, nous considérons les exigences posées par les méthodes d'estimation, non pas comme un effort mais plutôt comme un investissement récupérable dans la suite du développement de l'application, par exemple comme outil de documentation et de communication.

2. Notre modèle d'application.

2.1. Introduction.

Etant donné que la méthode des PF est utilisable à postériori, on peut dire que l'estimateur connaît le nombre exact d'entrées et de sorties de l'application, ainsi que leur composition en détails : par exemple, le nombre exact de type de records, de champs et de groupes de données logiques. C'est pour cette raison, à notre avis, qu'Albrecht ne voyait pas l'utilité de décomposer l'application en différents traitements afin d'en discerner leurs entrées et sorties respectives.

Dans les premières étapes du cycle de vie d'un projet, il est certain qu'on ne dispose pas de ces informations, mais il est possible de les approcher à un niveau logique lorsque l'analyse conceptuelle est terminée. En effet, à cette phase du développement dans la méthodologie de la C.G.E.R., les modèles

conceptuels des traitements et des données sont en principe réalisés. C'est à partir de ces informations qu'il faudra adapter la méthode des PF pour respecter notre objectif.

2.2. Les modifications

Pour rappel, dans le modèle de décomposition des traitements, on procède, par une approche top-down, à une réduction du problème global en une série de sous-problèmes moins complexes pour obtenir une arborescence. Chaque niveau de l'arborescence a un intérêt propre selon son degré de détail, il est donc utile par souci de standardisation de distinguer les différents niveaux comme suit : système - activités - tâches.

"Un système est un ensemble de traitements et de données fortement liés entre eux par des relations internes au système et pouvant être séparés de l'environnement du système tout en possédant certaines relations avec cet environnement" [Graas **].

"Par tâche, on entend un ensemble organisé d'opérations élémentaires, perçues et désignées globalement, manuelles et/ou automatisées. Ces opérations sont répétitives pour chaque échange d'informations transformant selon des règles prédéfinies des informations d'entrée en informations de sortie. Une tâche est, par ailleurs, caractérisée par son unicité de lieu, de temps et d'action" [Graas **].

La signification à attacher à ces trois types d'unicité a été précisée comme suit :

- "un traitement répond à une condition d'unicité dans le temps s'il effectue une séquence unique d'opérations sans interruption, séquence répétitive pour chaque échange d'information",
- "l'unicité de lieu implique que cette séquence est exécutée à un poste de travail",
- "l'unicité d'action caractérise un traitement qui utilise des règles fixes de transformation de l'information" [Delieux, Delvaux 87].

"Par activité, on entend un ensemble cohérent de tâches nécessaires à l'accomplissement d'une finalité permanente et essentielle du système" [Graas **].

Une fois la décomposition en traitements réalisées, on décrira une tâche par la technique du diagramme HIPO (cfr la définition de l'analyse conceptuelle dans la partie 1).

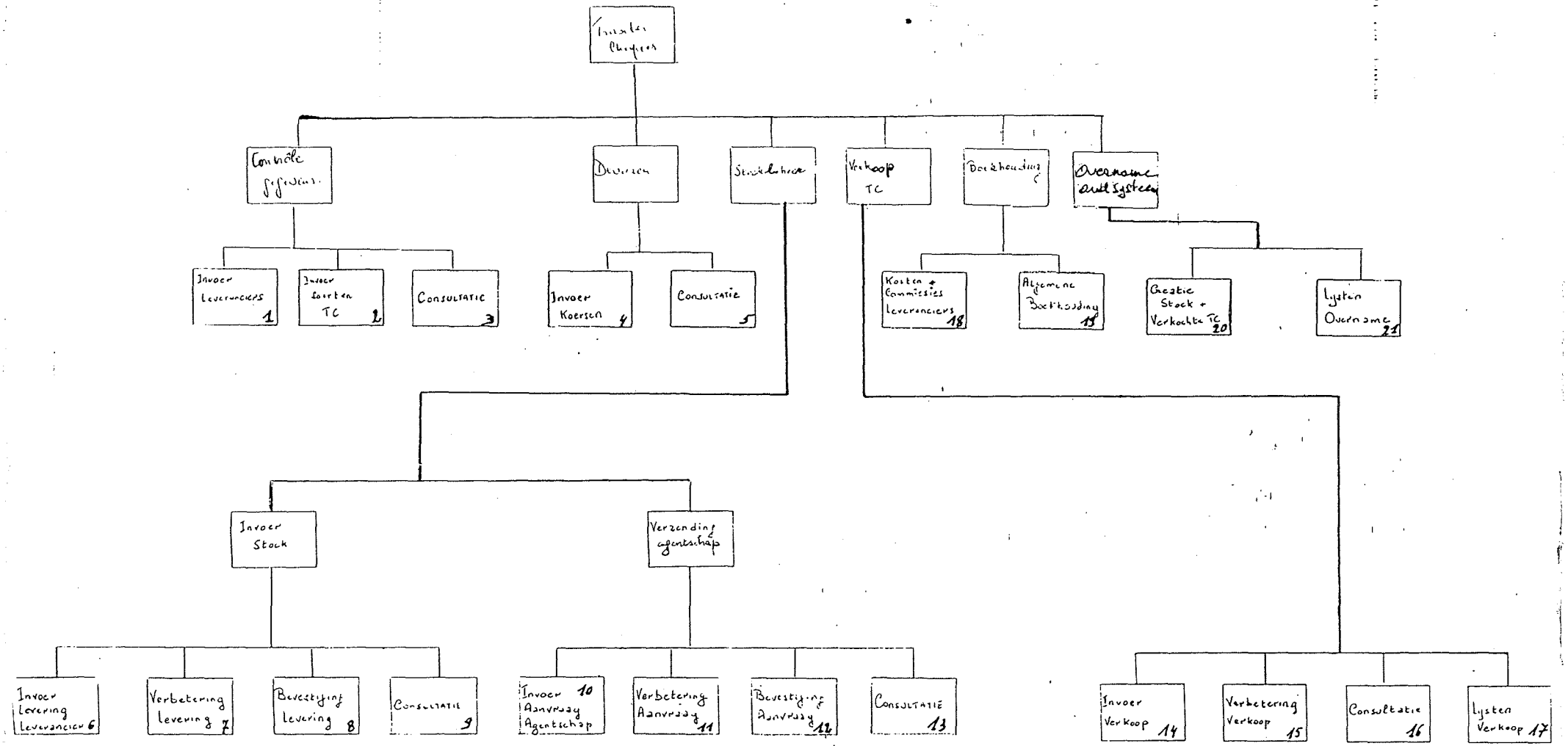
On disposera également à ce stade du modèle conceptuel des données sous la forme du modèle entité/relation.

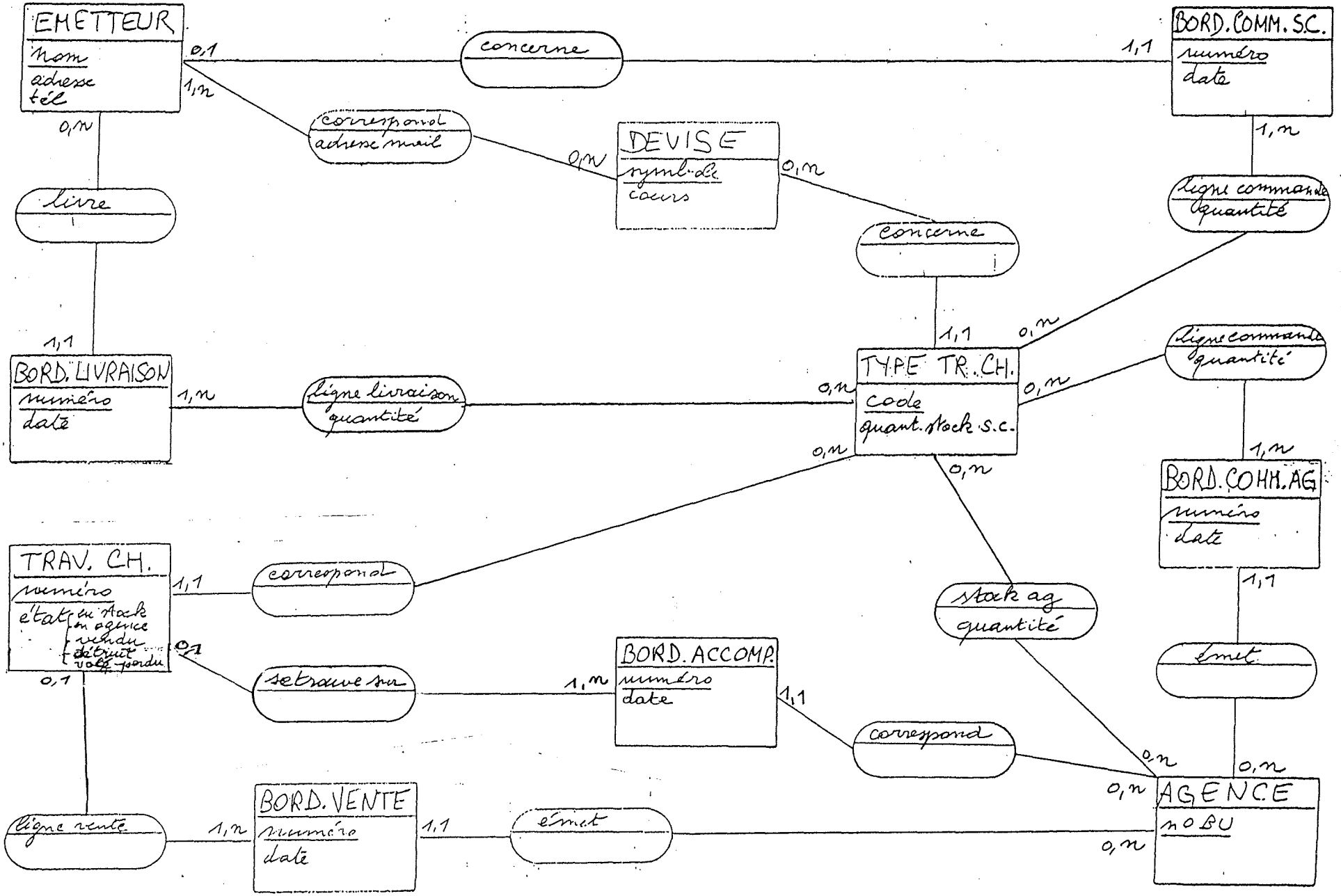
Désormais, nous utiliserons la méthode des PF au niveau de la tâche et non plus au niveau du système comme Albrecht le faisait. En effet, nous considérons qu'il est nécessaire par souci de précision de l'estimation, de connaître les entrées et sorties de chacune des tâches. Ainsi, le risque d'oublier une entrée ou une sortie semble fortement réduit.

Nous garderons les mêmes principes pour dénombrer les cinq types de fonctions définis dans Albrecht, mais au niveau de la tâche. Cependant, pour attribuer une complexité à ces fonctions, on parlera de sous-schémas de la base de données, de types d'entités et de types d'attributs en lieu et place de groupes de données logiques, de types de records et de champs. Nous garderons la notion de type de données incluant les attributs des entités et toute autre information nécessaire pour l'exécution de la tâche.

A titre d'illustration, la découpe en activités/tâches et le schéma entité/association de l'application TC sont donnés ci-après :

AKTIVITEITEN en TAKEN




EMETTEUR
nom
adresse
tél
concerne
BORD. COMM. S.C.
numéro
date
correspond
adresse mail
DEVISE
symbole
cours
livre
ligne commande
quantité
concerne
BORD. LIVRAISON
numéro
date
ligne livraison
quantité
TYPE TR. CH.
code
quant. stock S.C.
ligne commande
quantité
BORD. COMM. AG
numéro
date
TRAV. CH.
numéro
état
en stock
en agence
vendu
détruit
volé / perdu
correspond
stock ag
quantité
se trouve sur
BORD. ACCOMP.
numéro
date
émet
correspond
ligne vente
BORD. VENTE
numéro
date
émet
AGENCE
n° BU
0,1
1,n
0,n
1,1

Suivre une telle démarche constitue une première standardisation de la découpe sur laquelle sera basée l'estimation : deux estimateurs différents devraient pouvoir obtenir la même découpe et, de là, la même estimation. Ceci est cependant fort théorique car, en réalité, les gens interprètent de manière différente ce qu'il faut entendre par tâche. Nous pouvons illustrer ceci par l'exemple suivant : la tâche n°1 "invoer leveranciers" (introduction des émetteurs de TC) réalise l'ajout, la suppression et la modification d'émetteurs. Pourquoi ne pas en avoir fait trois tâches distinctes ?

Pour essayer de remédier à cette part de subjectivité dans la découpe, nous pourrions dire que nous avons obtenu un niveau de détails suffisamment grand si le critère suivant est respecté :

CRITERE :

On atteint un niveau de détails suffisant si on effectue une découpe du système en activités/tâches, et si pour chaque tâche, il est facile d'expliquer la production de son résultat à partir des différentes sources d'information nécessaires

Par sources d'information, il faut entendre :

- toutes les informations logiques susceptibles d'apporter un éclaircissement sur le fonctionnement de la tâche : par exemple, des règles de traitement,

- toutes les données qui seront lues ou modifiées afin d'obtenir le résultat attendu.

Si la découpe en activités/tâches du système a été réalisée et si toutes les formes d'information ont été dégagées pour chaque tâche, il devrait être possible de "démontrer" qu'à partir des entrées d'une tâche, on obtient le résultat. Cette démonstration devrait constituer le moyen de savoir si la découpe en tâches et l'analyse des I/O sont suffisamment détaillées.

EXEMPLE : supposons que le calcul du nombre de PF brut soit une des tâches d'une application. Selon notre critère, il faudrait la définir par les entrées suivantes :

- le nombre d'inputs, d'outputs, d'inquiries, etc... accompagnées d'une complexité simple, moyenne ou complexe,

- la table de conversion qui permet de traduire un niveau de complexité pour une des cinq fonctions données en un nombre de PF; par exemple, une input simple a comme valeur de conversion 3 PF,

- la formule de calcul des PF brut constitue un exemple de règle de traitement :

$$PFB = \sum_{j}^{5} \sum_{i}^{3} \text{nbre de fonctions}_{i,j} * \text{valconv}_{i,j}$$

avec i correspondant aux cinq types de fonctions, avec j correspondant aux trois niveaux de complexité et valconv étant la valeur de conversion.

Il faudrait également la définir par la sortie suivante :

- le nombre de PF brut.

On peut maintenant expliquer très simplement comment obtenir le nombre de PF brut à partir de ces entrées, puisqu'on dispose de la formule et de ses données nécessaires. Il faut remarquer que les entrées sont décrites à un niveau logique et qu'en réalité, certaines d'entre elles constitueront le code du programme, par exemple, les règles de traitement.

2.3. Les principes de fonctionnement du programme.

Nous avons implémenté la méthode des points de fonction sur un micro-ordinateur de façon à cacher le plus possible la "cuisine interne" du calcul du nombre de PF.

Nous demanderons que les étapes suivantes soient respectées pour pouvoir utiliser le programme :

Etape 1 : découper le système en tâches, tout en respectant le critère de découpe proposé.

Etape 2 : dénombrer les entrées et sorties de chaque tâche en les classant dans un des types suivants :

- Fichier de données (fd): il s'agit d'un fichier logique ou groupe d'entités comprenant les sources d'informations dans le sens défini ci-dessus.

- Ecran de données (ed): c'est un ensemble de types de données en entrée ou en sortie d'un terminal.

- Ecran de messages (em): ce sont toutes les entrées et sorties de messages contenant des informations autres que celles nécessaires à l'exécution d'une tâche (informations différentes des types de données).

- Document (d): c'est un ensemble de types de données en entrée ou en sortie placé sur un support physique.

Etape 3 : attribuer une complexité à chacune de ces entrées et sorties en utilisant les tableaux de complexité définis ci-dessous :

- **pour les fichiers de données :**

NBRE DE AT / NBRE DE ENT	1 à 19	20 à 50	51 et +
1	S	S	M
2 à 5	S	M	C
6 et +	M	C	C

AT : Attribut
ENT : Entité
S : Simple
M : Moyen
C : Complexe

TABLEAU 23

- **pour les écrans de données :**

NBRE DE TD / NBRE DE FD	1 à 4	5 à 15	16 et +
0 à 1	S	S	M
2	S	M	C
3 et +	M	C	C

TD : Type de données
FD : Fichier de données
S : Simple
M : Moyen
C : Complexe

TABLEAU 24

- **pour les documents :**

NBRE DE TD / NBRE DE FD	1 à 5	6 à 19	20 et +
0 à 1	S	S	M
2 à 3	S	M	C
4 et +	M	C	C

TD : Type de données
FD : Fichier de données
S : Simple
M : Moyen
C : Complexe

TABLEAU 25

- **pour les écrans de messages :** étant donné la diversité des types de messages, il nous semble difficile de construire un tableau de complexité. Nous suggérons d'attribuer un niveau de complexité en fonction du degré de difficulté pour produire cet écran. On peut toujours attribuer une complexité moyenne en cas de doute.

Etape 4 : distinguer les tâches qui consistent seulement en une simple consultation des données. (= requête)

Etape 5 : donner la chronologie des tâches (s'il en existe une), car il nous est nécessaire de connaître l'enchaînement des différentes tâches pour mener à bien le calcul du nombre de PF.

2.4. Le diagramme de flux.

Nous proposons que l'utilisateur fournisse les informations de ces cinq étapes par l'intermédiaire de la technique du diagramme de flux dont les règles de représentation et le formalisme sont les suivants.

2.4.1. Règles de représentation.

- un fichier modifié par la tâche doit se trouver aussi bien en entrée qu'en sortie de la tâche.

- un fichier créé par une tâche doit seulement se trouver en sortie de la tâche.

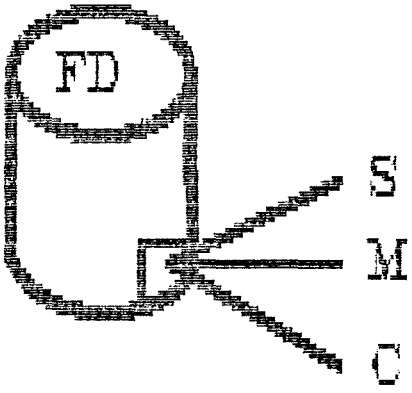
- un fichier consulté par une tâche doit seulement se trouver en entrée de la tâche.

- donner un nom identifiant à chaque écran, chaque document, chaque fichier et chaque tâche.

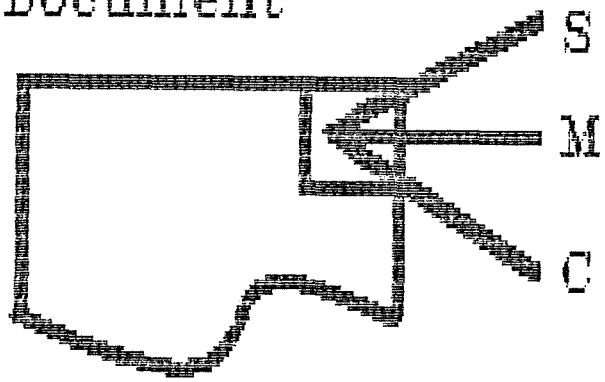
- attribuer une complexité (simple, moyenne ou complexe) à chaque fichier, chaque écran et chaque document.

2.4.2. Le formalisme du diagramme de flux.

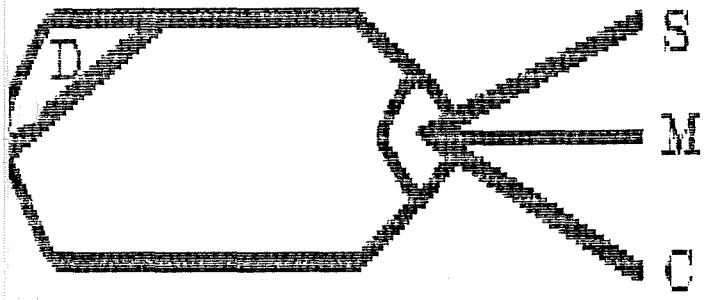
Fichier de données



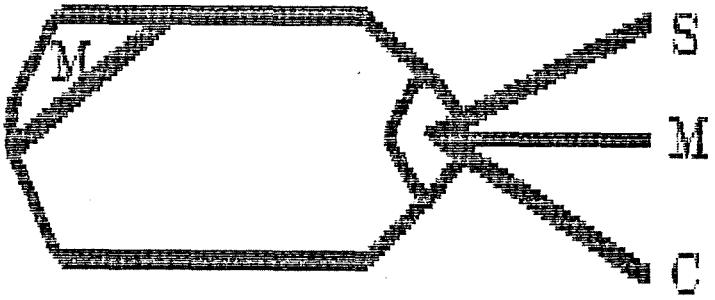
Document



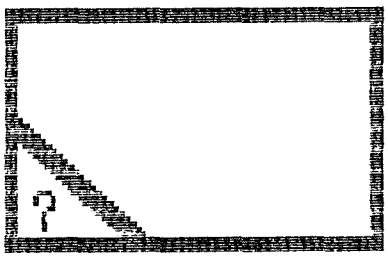
Ecran de données



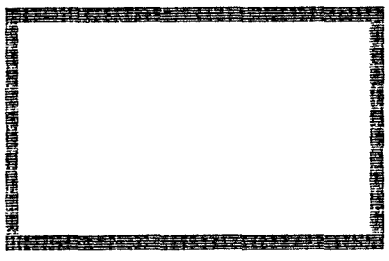
Ecran de messages



Tâche-Inquiry



Tâche sans Inquiry



A titre d'illustration, nous présentons, en annexe 1, le diagramme de flux réalisé pour l'application TC.

2.4.3. Les avantages du diagramme de flux.

L'adoption du diagramme de flux nous semble judicieuse à plus d'un titre :

- nous voulons cacher à l'utilisateur certaines notions relatives à la méthode des points de fonction. Nous voulons lui donner l'impression qu'en utilisant le diagramme de flux, il ne fait que décrire son projet, sans entrer dans la cuisine interne des PF.

- il permet une bonne communication avec l'utilisateur. Par conséquent, c'est un moyen de contrôler si toutes les fonctionnalités demandées par l'utilisateur sont respectées dans le projet.

- c'est un moyen de vérifier si le niveau de détails est suffisant (la somme des démonstrations des traitements doit prouver la satisfaction des fonctions demandées par l'utilisateur).

- il peut être aussi un outil de documentation utile à la maintenance.

- il est une bonne formalisation du flux de données traversant la frontière imaginaire de chaque tâche.

- il peut servir d'argument pour l'introduction d'une méthodologie de développement dans les milieux les plus réticents. En effet, l'estimation de la charge d'un projet constitue une application concrète de la construction d'un diagramme de flux.

- il pourrait constituer l'entrée de notre programme si une interface graphique était réalisée.

2.5. La grammaire du programme.

Etant donné les définitions des cinq fonctions de la théorie d'Albrecht et le formalisme de notre diagramme de flux, nous avons dégagé une grammaire selon laquelle le programme classifie les données du diagramme de flux en inputs, outputs, inquiries, master files et interfaces.

Nous avons défini la grammaire de la manière suivante :

si pour la j*** tâche :

<ed$_j$.E> = un écran de données en entrée de la j*** tâche

$<ed_j.S>$ = un écran de données en sortie de la j*** tâche

$<d_j.E>$ = un document en entrée de la j*** tâche

$<d_j.S>$ = un document en sortie de la j*** tâche

$<fd_{ij}.E>$ = i*** fichier de données en entrée de la j*** tâche

$<fd_{ij}.S>$ = i*** fichier de données en sortie de la j*** tâche

$<em_{ij}.E>$ = i*** écran de messages en entrée de la j*** tâche

$<em_{ij}.S>$ = i*** écran de messages en sortie de la j*** tâche

et si

"::=" signifie "est défini par",

"||" représente le "ou" logique,

"|" représente le "et" logique,

"-" représente le "non" logique,

alors, nous pouvons redéfinir les cinq fonctions de la manière suivante :

input ::= $<fd_{ij}.E>$ | - $<fd_{ij}.S>$ || $<ed_j.E>$ || $<em_{ij}.E>$||

$<d_j.E>$

output ::= $<em_{ij}.S>$ | $<em_{ik}.E>$ || $<ed_j.S>$ || $<d_j.S>$

avec k>j

master file ::= $<fd_{ij}.S>$ | $<fd_{ij}.E>$

inquiry ::= $<ed_j.E>$ | $<ed_j.S>$ || $<ed_j.E>$ | $<d_j.S>$

interface ::= $<fd_{ij}.S>$ | - $<fd_{ij}.E>$ || $<fd_{ij}.E>$ |

$<fd_{ik}.S>$| - $<fd_{ik}.E>$ avec k<j.

Exemples de lecture de la grammaire :

Une output pour la tâche j est soit un document en sortie de cette tâche, soit un écran de données en sortie, soit un écran de message i en sortie de la tâche j et en entrée d'une tâche k, avec k chronologiquement supérieur à j. D'après cette illustration, on peut remarquer que la chronologie des tâches que nous demandons est indispensable pour réaliser notre analyse.

Un autre exemple : pour qu'un fichier de données i soit une interface pour la tâche j, il doit être en entrée de la tâche j et en sortie d'une tâche k mais ne doit pas être en entrée de cette tâche k, avec k inférieur à j.

2.6. Remarques.

Dans notre grammaire, la notion d'interface est simplifiée par rapport à celle présentée dans la théorie originale, car nous n'envisageons pas les relations inter-applications. Nous nous en sommes tenus à une relation inter-tâches.

Le modèle tel que nous l'avons pensé, s'adapte bien pour réaliser une estimation en fin d'analyse conceptuelle. Cependant, à ce moment, le développement du projet est déjà bien avancé. Notre modèle d'estimation, par conséquent, perd un peu son côté prévisionnel. Serait-il alors possible de l'utiliser en fin de définition de projet ?

Bien que les modèles des données et des traitements ne soient pas disponibles, il est néanmoins possible de dégager quelles sont les fonctionnalités du programme et les informations nécessaires pour les exécuter. Celles-ci pourraient être représentées sous la forme du diagramme de flux tel que nous l'avons défini dans notre modèle. Elles pourraient ainsi servir de base à une première estimation.

Cette estimation serait bien sûr moins précise que celle réalisée en fin d'analyse conceptuelle, mais elle pourrait être utile pour comparer des solutions alternatives en fonction de leur poids exprimé en PF. Nous pouvons ajouter que cette idée rejoint la proposition du MAI de s'orienter vers la production d'un diagramme de flux comme output de la définition de projet. De cette manière, ce diagramme de flux serait non seulement l'entrée de notre programme d'estimation, mais aussi un moyen de simplifier la phase de définition de projet qui jusqu'à présent se compose de nombreux documents fatidieux à compléter.

2.7. Mode d'emploi du programme.

On peut trouver en annexe 2 le mode d'emploi du programme ainsi que les résultats de l'estimation de l'application TC.

3. Estimation de l'effort et de la productivité à partir d'un nombre de PF.

3.1. Une première estimation.

Nous sommes arrivés à estimer la valeur fonctionnelle d'un logiciel exprimée en un nombre de PF. De cette manière, nous avons répondu au souhait de la C.G.E.R. qui consistait seulement à comparer des solutions en fonction de leur poids respectif. Mais en ce qui nous concerne, nous aimerions estimer l'effort de développement à partir de cette taille.

Albrecht et Gaffney en 83 ont déjà proposé un ensemble de formules permettant de traduire un nombre de PF en un effort [Albrecht, Gaffney 83] :

w = l'estimation de l'effort
= (54 * PF) - 13390

Cependant, on peut dire que cette formule pose un problème : un projet n'atteignant pas 248 PF demanderait un effort de développement négatif. Encore une fois, on voit très bien l'influence de l'environnement qui a agi sur la détermination de cette formule. En effet, ces constantes ont été obtenues par calculs statistiques.

C'est pourquoi, nous pensons qu'il faut éviter des formules toutes faites imprégnées de leur environnement d'expérimentation et qu'il faut par conséquent réaliser un suivi des projets développés au sein de sa propre organisation comme Behrens le propose dans son article [Behrens 83].

Pour rappel, l'effort n'est jamais que la taille divisée par la productivité. Il nous reste donc à calculer l'inconnue : la productivité.

La meilleure manière pour approcher cette inconnue, c'est de déterminer une productivité moyenne pour un environnement particulier. Pour ce faire, il faudrait enregistrer pour chacun des projets le nombre de points de fonction calculé à priori et, une fois le projet terminé, enregistrer le nombre d'heures consacré à son développement. A partir de ces données, on pourrait calculer la productivité moyenne pour des projets de différentes tailles, comme le montre le schéma de Behrens en figure 35.

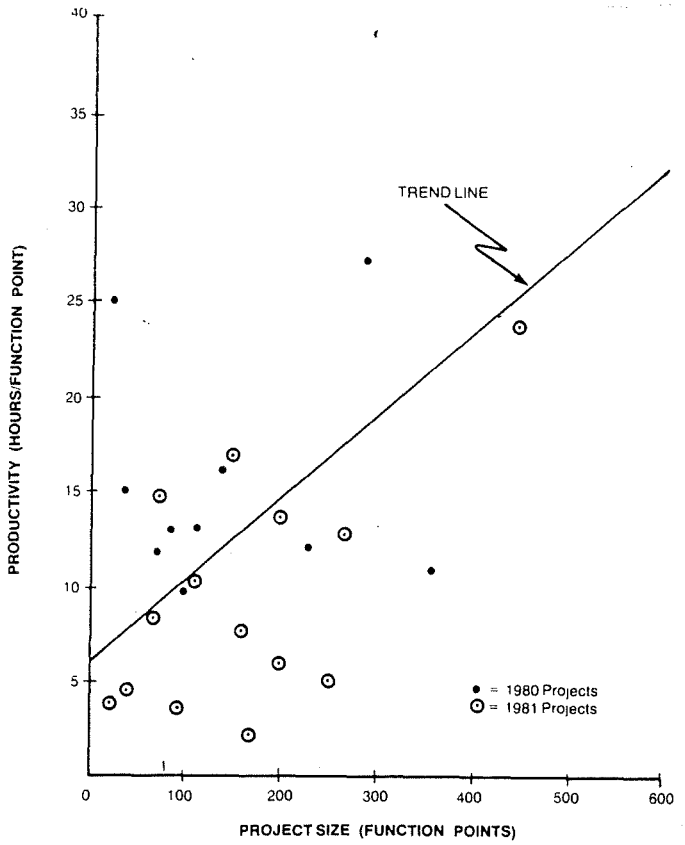


FIGURE 35 INFLUENCE DE LA TAILLE
D'UN PROJET SUR LA PRODUCTIVITE

Behrens y a représenté chaque projet par sa taille exprimée en PF et par sa productivité. Il a réalisé un suivi de ses applications pendant deux années et de cette manière, il a pu dégager la relation linéaire existante entre la taille et la productivité.

Par conséquent, pour un projet d'une certaine taille exprimée en PF, on peut dégager sa productivité attendue : celle qui correspond à la droite. Il suffit ensuite d'estimer la durée de développement par un simple calcul :

nombre de PF / productivité attendue = nombre d'heures

Par exemple, si on a estimé la taille d'un logiciel à 100 PF et si, en fonction de la droite (de Behrens), sa productivité attendue est de 5 PF par heure, alors la durée de développement est estimée à 20 heures. Si on se base sur la constatation de Boehm qu'un M-H représente 152 heures de travail, on pourrait traduire la durée de développement en un effort exprimé en M-H, c'est-à-dire 20/152 M-H.

3.2. Raffinement des estimations.

A postériori, il est bien entendu possible de vérifier la validité de la productivité qui intervient dans l'estimation de l'effort à priori. Chaque fois que l'on estime un écart de productivité entre l'estimée et la réelle, il faudrait réaliser une étude pour en déterminer les causes, c'est-à-dire les facteurs de coût. Ceci demande que l'on enregistre bien plus d'informations qu'une taille en PF et une durée de développement. Il faudrait enregistrer toutes les caractéristiques des projets pour pouvoir dégager les causes des écarts de productivité.

En effet, puisqu'on a calculé une productivité moyenne sur une période suffisamment longue, on peut dire que tous les facteurs ayant une influence sur la productivité sont implicitement compris dans cette productivité moyenne. Par conséquent, si un projet s'écarte de la productivité moyenne, cela signifie qu'il présente soit des caractéristiques différentes de celles des autres applications, soit des caractéristiques similaires mais d'une influence non habituelle.

C'est par l'étude de ces caractéristiques, que l'on pourrait établir des multiplicateurs d'effort propres à l'organisation. Par conséquent, pour des estimations ultérieures, on corrigerait la productivité moyenne par ces multiplicateurs si les caractéristiques des projets présentaient des influences sortant de la moyenne. Cependant, de telles études requièrent un investissement très important.

Conclusion

Il n'existe pas une méthode suffisamment générale pour s'adapter à tout environnement sans procéder à une nouvelle calibration du modèle. Ceci est dû à la démarche elle-même des méthodes d'estimation. En effet, elles procèdent souvent par essai et erreur, pendant une période plus ou moins longue, pour dégager un "comportement" moyen des logiciels analysés afin d'estimer celui des projets à venir. Pour corriger cette moyenne, les auteurs proposent généralement un certain nombre de facteurs qui sont censés tenir compte du contexte particulier de chaque développement, c'est par exemple la constante "état de la technologie" de Putnam, les 15 "cost drivers" de Boehm ou encore les 14 caractéristiques d'Albrecht. Ces facteurs sont issus de l'environnement d'expérimentation. Ce sont ceux qui ont une influence significative sur l'effort dans cet environnement, mais rien n'indique que ce sont les mêmes qui seront retenus dans un autre environnement. Cette diversité de facteurs a été longuement illustrée dans la partie 1 du travail.

Nous avions déjà évoqué à plusieurs reprises ce problème de la dépendance des méthodes par rapport à leur environnement d'expérimentation, mais étant donné l'importance considérable de ce problème, nous ne pouvions imaginer de conclure ce travail sans en reparler.

Un deuxième point sur lequel nous voudrions insister est la façon dont il faudrait considérer les méthodes d'estimation : en effet, nous pensons que beaucoup de personnes attendent beaucoup trop de ces méthodes, elles exigent souvent un résultat très précis sans y apporter un effort important. Les partisans de cette idée seront sûrement déçus, du moins dans l'état actuel des choses. En ce qui nous concerne, nous les voyons plus comme un outil d'aide à une estimation, comme un moyen de baser une estimation sur des critères concrets et non plus sur une intuition qui par définition est difficilement justifiable. Il est plus que nécessaire de revoir son jugement quant à l'utilisation des méthodes. En effet, comme nous l'avions déjà suggéré au cours de ce travail, nous pensons qu'une méthode d'estimation est un bon complément à une méthodologie de développement. Le modèle d'estimation tel que nous l'avons conçu dans l'adaptation des points de fonction par le diagramme de flux, peut servir de base à une bonne documentation du projet et être utile pour réaliser une définition de projet précise. De cette manière, les développeurs réticents à toute méthodologie pourraient voir le diagramme de flux comme une technique utile car il est la base de notre modèle d'estimation.

L'objectif que nous nous étions assignés, à savoir l'attribution à priori d'un poids à un nouveau projet, est satisfait du moins théoriquement par l'adaptation de la méthode des PF que nous avons réalisée. Pour arriver à ce résultat, cela nous a demandé de comprendre dans un premier temps tous les paramètres intervenant dans une estimation, et particulièrement ceux qui guident la méthode des PF. Etant donné le manque de clarté de cette dernière, nous avons été obligés d'élaborer un modèle d'application sur lequel, à notre avis, la méthode repose. Après l'avoir de cette manière mieux cernée, nous avons étudié dans quelle mesure elle était

transposable à priori. En effet, la connaissance des fonctionnalités d'un projet dès le début du développement permet cette transposition.

Seulement, dans l'état actuel des choses, nous n'obtenons qu'un poids qui peut être utile dans une comparaison de solutions possibles pour un projet. Mais si on veut obtenir plus d'informations prévisionnelles telles que la durée ou l'effort de développement, cela demande une utilisation à long terme de la méthode pour obtenir une conversion du nombre de PF en un nombre d'heures. Nous obtiendrions ainsi un nombre de PF par heure, c'est-à-dire une productivité moyenne. Cependant, la justesse de ces estimations dépendra de l'investissement en temps et en effort que l'on voudra bien y consentir : la productivité moyenne n'est que la base de l'estimation et en cas d'échec, il faudra analyser les écarts de productivité pour déterminer les facteurs qui en sont la cause et ainsi améliorer les estimations futures en établissant des multiplicateurs d'effort. On peut imaginer le travail que requièrent de telles analyses.

Pour obtenir la productivité moyenne, nous aurions souhaité pouvoir exécuter notre programme pour plusieurs applications mais, étant donné les difficultés rencontrées pour l'étude d'une application simple (Travellers Cheques) et compte tenu de la limite en temps et de la confidentialité des projets, il paraissait irréalisable d'aborder d'autres applications. Ce serait donc une étude à mener.

Nous pouvons dire également que ce travail nous a apporté énormément, en ce sens que nous n'avions jamais pour ainsi dire abordé l'aspect économique du développement d'un logiciel. Nous avons pu approché tous les problèmes sous-jacents, et surtout nous avons pu apprécier l'importance pour toute organisation de connaître les coûts liés au développement.

Bibliographie

[Abdel-Hamid 86]

Abdel-Hamid, T.K., "Impact of schedule estimation on software project behavior", IEEE software, July 1986.

[Albrecht 79]

Albrecht, A.J., "Measuring application development productivity", The proceedings of the joint Share/Guide/IBM application development symposium, oct. 1979.

[Albrecht, Gaffney 83]

Albrecht, A.J., Gaffney, J.E., "Software function, source lines of code and development effort prediction : a software science validation", IEEE transactions on software engineering, vol. SE-9, june, 1983.

[Audit **]

Audit Informatica B.V. "Handboek : functie punt analyse", Almere, 1986.

[Bailey 81]

Bailey, J.W., "A meta model for software development resource expenditures", Proceedings 5TH International conference on software engineering, IEEE catalog n°CH1627-9, 1981.

[Basili 80]

Basili, V.R., "Tutorial on models and metrics for software management and engineering", "Product metrics", IEEE catalog EHO-167-7, IEEE computer society, 1980.

[Behrens 83]

Behrens, C., "Measuring the productivity of computer systems development activities with function points", IEEE,1983.

[Bemelmans 81]

Bemelmans, Th.M.A., de Boer, J.G., "Ontwikkelingsmethoden I : het ontwikkeling van informatiesystemen", Informatie, jaargang 23, n°1, feb. 1981.

[Bodart 83]

Bodart, F., Pigneur, Y., "Conception assistée des applications informatiques, 1. Etude d'opportunité et analyse conceptuelle", Masson, Presses universitaires de Namur, 1983.

[Boehm 81]

Boehm, B.W., "Software engineering economics", Prentice Hall, 1981.

[Boehm 83]

Boehm, B.W., "Seven principles of software engineering", The journal of systems on software, n°3, p. 3, 1983.

[Boehm 84]

Boehm, B.W., "Software engineering economics", IEEE Transactions on software engineering, vol.-SE-10, n°1, jan 1984.

[Boehm, Wolverton 80]

Boehm, B.W., Wolverton, R.W., "Software cost modelling : some lessons learned", The journal of systems and software, 1980.

[Brooks 75]

Brooks, F.P., "The mythical man-month", Addison-Wesley, reading MA 1975.

[Christensens, Fitsos, Smith 81]

Christensen, K., Fitsos, G.P., Smith, C.P., "A perspective on software science", IBM system journal, vol. 20, n°4, 1981.

[Conté 86]

Conté, S.D., Dunsmore, H.E., Shen, V.Y., Thebaut, S.M., "Cost estimation for software development", onderzoeksvoorstellen van het center for software engineering research (CSER), 1986.

Conté, S.D., Dunsmore, H.E., Shen, V.Y., "Software engineering metrics and models", Benjamin Cummings, 1986.

[Craenen 84]

Craenen, G., "Begroten van automatiseringsprojecten", Informatie, jaargang 26, n°3, p.173, t/m 268, mars 1984.

[de Kater 87]

de Kater, Al, Beleids informatica tijdschrift, vol. 11, n°2, tweede kwartier 1985.

[Delieux, Delvaux 87]

Delieux, N., Delvaux, B., "Etude d'une méthodologie et d'outils d'aide au développement d'application de gestion", FNDP Namur, 1987.

[DeMarco 82]

DeMarco, T., "Contolling software projects", Yourdon Press, 1982.

[Fairley 87]

Fairley, R., "Software engineering concepts", McGraw Hill, 1987.

[Graas **].

Graas, C., "Introduction à la méthodologie", cours MAI, C.G.E.R., Bruxelles juin 1985.

Graas, C., "Définition de projet", cours MAI, C.G.E.R., Bruxelles juin 1986.

[Green 85]

Green, J., "Er is dringend behoefte aan meetbare programmatuurproductie", Supplement computable, 10 mei 1985.

[Halstead 77]

Halstead, Maurice H., "Elements of software science", Elsevier North Holland, inc., 1977.

[Heemstra 87]

Heemstra, F.J., "Wat bepaalt de Kosten van software ?", Informatie, jaargang 29, extra editie, 1987.

[Herrman 84]

Herrman, O., "Analyse der Einfluszfaktoren auf die kosten van softwareentwicklungen", Angewandte informatik, n°4, 1984.

[Jeffery 85]

Jeffery, D.R., Lawrence, M.J., "Managing programming productivity", The journal of systems on software, n°5, 1985.

[Jones 84]

Jones, C., "Reusability in programming : a survey of the state of the art", IEEE transactions on software engineering, vol. SE-10, n°5, sept. 1984.

[Jones 86]

Jones, C., "Programming productivity", MC Graw Hill, 1986.

[Kemerer 87].

Kemerer, C.F., "An empirical validation of software cost estimation models", Communications of the ACM, 30, may 1987.

[Kitchenham 85]

Kitchenham, B.A., "Software project development cost estimation", journal of systems and software, may 1985.

[Magérat 85]

Magérat, Gérard, "Les projets de développement informatique : estimation de leur charge de travail et mesure de leur productivité", Crédit Communal de Belgique, sept. 1985.

[MAI **]

M.A.I., "Analyse conceptuelle des données", version 1.0, C.G.E.R., Bruxelles.

M.A.I.,"Aperçu de la filière de développement de projets informatiques", version 1.0, C.G.E.R., Bruxelles.

[Martin 83]

Martin, J., McClure, C., "Software maintenance : the problems and its solutions", Prentice Hall Incorporation, Englewood Cliffs, N.J., 1984.

[Mohanty 79]

Mohanty, Siba N., "Models and measurements for quality assessment of software", Computing surveys, vol. 11, 1979.

[Nanus 64]

Nanus, Burt, "Some cost contributors to large-scale programs", proceedings - spring joint computer conference, 1964.

[Nelson 66]

Nelson, E.A., "Management handbook for the estimation of computer programming costs", AD-A648750, systems development corp., 1966.

[Parkinson 57]

Parkinson, G.N., "Parkinson's law and other studies in administration", Houghton-Mifflin, Boston, 1957.

[Parr 80]

Parr, F.N., "An alternative to the Rayleigh curve model for software development effort", IEEE transactions on software engineering, SE-6, mars 1980.

[Perry 83]

Perry, William E., "Effective methods of EDP quality assurance", Prentice Hall, 1983.

[Putnam 78]

Putnam, L.H., "A general empirical solution to the macro software sizing and estimating problem", IEEE transactions on software engineering,p. 345-361, july 1978.

[Putnam 79]

Putnam, L.H., Fitzsimmons, A.,"Estimating software costs", Datamation, sept., oct., nov., 1979..

[Rudolph 83]

Rudolph, E.E., "Productivity in computer application development", Department of mamagement studies, University of Auckland, 1983.

[Rubin 85]

Rubin, H.A., "A comparison of cost estimation tools (a panel discussion)", Proceedings 8TH International conference on software engineering, IEEE, 1985.

[Surböck 78]

Surböck, E.K., "Management von EDV projekten", Berlin, 1978.

[Thibodeau 81]

Thibodeau, R., "An evaluation of software cost estimating models", RADC-TR-81-144, 1981.

[van Vliet 87]

van Vliet, J.C., "Wat kost dat nou, zo'n programma ?", Informatie, jaargang 29, extra editie, 1987.

[Walston 77]

Walston, C.E., Felix, C.P., "A method of programming measurement and estimating", IBM systems journal 16, 1977.

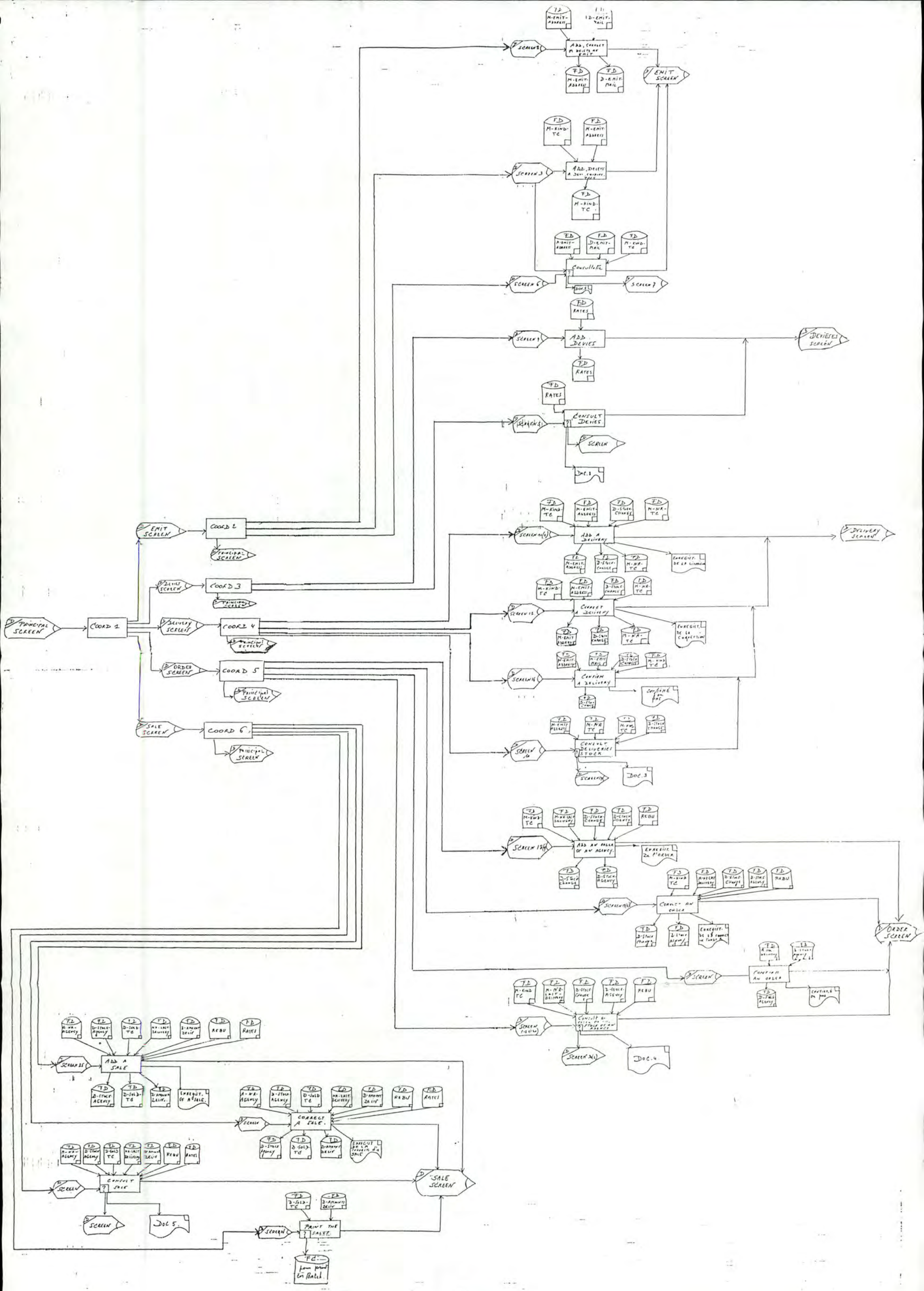
[Wolverton 74]

Wolverton, R.W., "The cost of developing large-scale software", IEEE transactions on computers, 1974.

[Wang 84]

Wang, A.S., "The estimation of software science and effort : an approach based on the evolution of software metrics", PhD thesis, department of computer science, Purdue University, 1984.

Annexe 1


PRINCIPAL SCREEN
COORD 1
EMIT SCREEN
COORD 2
PRINCIPAL SCREEN
DELIVERY SCREEN
COORD 3
PRINCIPAL SCREEN
DELIVERY SCREEN
COORD 4
PRINCIPAL SCREEN
ORDER SCREEN
COORD 5
PRINCIPAL SCREEN
SALE SCREEN
COORD 6
PRINCIPAL SCREEN
SCREEN 1
SCREEN 2
SCREEN 3
SCREEN 4
SCREEN 5
SCREEN 6
SCREEN 7
SCREEN 8
SCREEN 9
SCREEN 10
SCREEN 11
SCREEN 12
SCREEN 13
SCREEN 14
SCREEN 15
SCREEN 16
SCREEN 17
SCREEN 18
SCREEN 19
SCREEN 20
SCREEN 21
SCREEN 22
SCREEN 23
SCREEN 24
SCREEN 25
ADD, CORRECT OR DELETE AN EMIT
ADD, DELETE A DELIVERY COMPANY TYPE
CONSULT A DELIVERY
ADD DEVISES
CONSULT DEVISES
ADD A DELIVERY
CORRECT A DELIVERY
CONFIRM A DELIVERY
CONSULT DELIVERIES STOCK
ADD AN AGENT OF AN AGENCY
CORRECT AN ORDER
CONFIRM AN ORDER
CONSULT AN ORDER
ADD A SALE
CORRECT A SALE
CONSULT SALE
PRINT THE SALES
DEVISES SCREEN
EMIT SCREEN
DELIVERY SCREEN
ORDER SCREEN
SALE SCREEN
SCREEN
DOC 1
DOC 2
DOC 3
DOC 4
DOC 5
FD M-EMIT-AGENCY
FD D-EMIT-MAIL
FD M-KIND-TC
FD M-NR-TC
FD D-STOCK-CHANGE
FD REBU
FD RATES
FD D-STOCK-AGENCY
FD D-SOLD-TC
FD D-AMOUNT-DELIV
FD M-NR-DELIVERY
FD M-EMIT-MAIL
FD D-STOCK-AGENCY
ENREGIST. DE LA LIVRAISON
ENREGIST. DE LA CORRECTION
CONFIRMÉ PAT
ENREGIST. DE LA COMMANDE
ENREGIST. DE LA VENTE
FC: Lancement en Batch.

RESULTATS DE LA METHODE DES POINTS DE FONCTION

NOM DU PROJET : travellercheq

LE NOMBRE DE POINTS DE FONCTION BRUT = 663

LE NOMBRE DE POINTS DE FONCTION NET = 663

POUR : 71 INPUTS
51 OUTPUTS
22 MASTER FILES
5 INQUIRIES
0 INTERFACES

ET AVEC UNE CORRECTION DE : ± 0 %

TAPEZ UNE TOUCHE

Annexe 2

I. La philosophie du programme.

II. Les différentes fonctions du programme.

1. L'estimation du coût d'un projet.

1.1. La création d'un nouveau projet.

1.2. La consultation, la modification ou l'estimation d'un ancien projet.

1.2.1. La consultation des fiches I/O.

1.2.2. La consultation des caractéristiques des I/O.

1.2.3. La modification des traitements et des I/O.

1.2.3.1. La correction des caractéristiques des I/O du projet.

1.2.3.2. L'ajout d'I/O dans le projet.

1.2.3.3. L'ajout d'I/O dans un traitement.

1.2.3.4. La suppression d'I/O dans le projet.

1.2.3.5. La suppression d'I/O dans un traitement.

1.2.3.6. L'inversion d'I/O dans un traitement.

1.2.3.7. L'ajout d'un traitement dans le projet.

1.2.3.8. La suppression d'un traitement dans le projet.

1.2.4. Le lancement du calcul d'estimation.

2. La modification des paramètres de la méthode d'estimation.

3. L'historique des estimations (non encore développé).

1. La philosophie du programme.

Nous avons voulu cacher à l'utilisateur les mécanismes de calcul de la méthode des points de fonction. C'est pourquoi, il est nécessaire d'établir au préalable un diagramme de flux décrivant chronologiquement les traitements du projet par leurs entrées et sorties (fichiers, documents, écrans ...). Une fois ce diagramme de flux introduit dans le programme, ce dernier calculera à l'aide de sa grammaire interne le nombre de points de fonction en dénombrant les inputs, les outputs, les interfaces, les inquiries et les master files parmi les entrées et sorties des traitements du projet.

II. Les fonctions du programme.

1. L'estimation du coût d'un projet.

1.1. La création d'un nouveau projet.

Après avoir sélectionner l'option 1 de l'écran 1 (menu principal), l'écran 2 affiche les noms des projets existants déjà dans les fichiers du programme. Pour créer un nouveau projet, l'utilisateur doit introduire un nom de projet différent des noms de projet existants déjà.

REM : il est impossible d'appeler deux projets différents par le même nom, car le nom d'un projet est identifiant dans le programme.

Le nom du nouveau projet introduit, l'écran 3 présente deux options : continuer la création du nouveau projet ou retourner au menu principal.

En choisissant l'option 1, l'utilisateur déclenche l'écran 4 pour l'introduction des caractéristiques de chaque i/o du projet ayant un nom identifiant.

De cette manière, pour décrire un traitement par ses entrées et sorties, l'utilisateur n'aura plus qu'à donner le nom d'une i/o sans spécifier son type et sa complexité car ceux-ci sont déjà connus .

L'introduction d'une i/o s'effectue de la manière suivante :

a) l'utilisateur entre le type de l'i/o,

à savoir :

fd : pour un fichier de données
ed : pour un écran de données
em : pour un écran de messages

d : pour un document
?ed : pour un écran de données déclenchant ou résultant d'une requête
?d : pour un document résultant d'une requête

b) ensuite le nom de l'i/o

REM : le nom d'une i/o est identifiant dans le projet (impossible donc de trouver deux i/o de même nom)

c) enfin la complexité de l'i/o.

Pour quitter la procédure d'introduction des i/o, il faut taper <return> quand le curseur revient dans la zone d'introduction du type.

Alors, s'affiche l'écran 29 représentant une fiche de traitement vierge à remplir. Pour remplir cette fiche, il faut :

- taper un nom de traitement identifiant

  REM : comme pour les noms des projets et des i/o, le nom d'un traitement est identifiant dans le projet.

- décrire quelles sont les entrées et les sorties de ce traitement :

  . taper un nom d'i/o connu par le programme c'est-à-dire introduite dans la procédure précédente (cfr les points a)b)c) ci-dessus)

REM : si l'utilisateur désire revoir toutes les i/o qu'il a introduites dans la procédure précédente, il peut les faire défiler dans une fenêtre en tapant < ? return > dans la zone de l'écran "nom de l'entrée ou de la sortie ". Il peut alors faire défiler la fenêtre par les touches de fonction F1 et F2 ou quitter celle-ci par la touche de fonction F3.

. taper <E> pour signaler que cette i/o est entrée du traitement ou <S> si elle en est sortie.

REM : il est impossible d'introduire une i/o plusieurs fois en entrée ou en sortie du même traitement.

Pour quitter cette fiche et passer à la création du traitement suivant, taper <return> dans la zone " nom d'une entrée ou d'une sortie " ; apparaît alors une nouvelle fiche vierge.

Pour quitter la création des traitements, taper <return> dans la zone " nom du traitement " d'une fiche vierge. Ceci provoque le retour à l'écran 6 et le projet est considéré comme ancien projet.

1.2. La consultation, la modification ou l'estimation d'un ancien projet.

Pour pouvoir employer ces fonctions, il faut soit que l'utilisateur ait introduit un nom de projet existant dans l'écran 2, soit qu'il ait terminé la création d'un nouveau projet. Dans ces deux cas, l'écran 6 affiche un menu de 5 options.

1.2.1. La consultation des fiches i/o .

Sélectionner l'option 1 de l'écran 6 pour que l'écran 7 présente les numéros et les noms des traitements du projet.

Taper un numéro correspondant à un traitement (ou 0 pour sortir de la consultation) pour que l'écran 8 présente le traitement choisi.

Il est possible de visualiser les traitements précédents ou suivants en tapant <-> ou <+>. En tapant <S>, on sort du dernier traitement consulté avec un retour vers l'écran7.

1.2.2. Consultation des caractéristiques des i/o.

Sélectionner l'option 2 de l'écran 6 pour que l'écran 9 affiche l'ensemble des i/o du projet. Comme dans la consultation précédente, les mêmes fonctions de consultation sont offertes à l'utilisateur.

1.2.3. Modification des traitements et des i/o.

Sélectionner l'option 3 de l'écran 6 et l'écran 10 affiche 9 sous-options :

1.2.3.1. Correction des caractéristiques des i/o du projet.

Sélectionner l'option 1 de l'écran 10.

L'écran 11 affiche la première page des i/o du projet. Les fonctions de consultation du point 1.2.2. sont toujours possibles.

Pour corriger une i/o, taper <C> et l'écran 12 vous demandera le numéro de l'i/o à corriger. Une fois ce numéro introduit, l'écran 13 fait clignoter en gras l'i/o à corriger. Pour corriger cette i/o, l'utilisateur doit en donner son type, son nom (toujours identifiant) et sa complexité; ces caractéristiques sont introduites de la même manière qu'au point 1.1. a)b)c) .

Ensuite l'écran 11 affiche la modification enregistrée et offre à nouveau les mêmes fonctions de consultation du point 2.2.

1.2.3.2. Ajout d'une i/o dans le projet.

Sélectionner l'option 2 de l'écran 10.

L'écran 14 affiche la première page des i/o du projet. Les fonctions de consultation du point 1.2.2. sont toujours possibles.

Pour ajouter une i/o, taper <A> et l'écran 15 affiche la dernière page d'i/o pour y ajouter une nouvelle à la suite des autres.
Les caractéristiques de cette nouvelle i/o sont introduites de la même façon qu'au point 1.1. a)b)c) .

1.2.3.3. Ajout d'une i/o dans un traitement.

Sélectionner l'option 3 de l'écran 10.
L'écran 16 présente les numéros et les noms des traitements. En choisissant un de ces numéros, l'écran 17 affichera le traitement associé à ce numéro. Les fonctions de consultation du point 1.1. sont toujours possibles. Pour ajouter une i/o, taper <A> et l'écran 18 vous demandera un nom d'i/o et où il faut l'ajouter, c'est-à-dire en entrée ou en sortie.

REM : le mécanisme de la fenêtre exposé ci-dessus est toujours applicable.
Il est impossible d'ajouter une i/o en entrée d'un traitement si elle s'y trouve déjà (pas de double). Ceci est également vrai du côté sortie d'un traitement.

1.2.3.4. Suppression d'une i/o dans le projet.

Sélectionner l'option 4 de l'écran 10.
Suivre la même procédure qu'au point 1.2.3.1. en suivant l'enchaînement des écrans 20 et 21.

1.2.3.5. Suppression d'une i/o dans un traitement.

Sélectionner l'option 5 de l'écran 10.
L'écran 22 donne les numéros et les noms des traitements. Choisir le numéro du traitement auquel il faut supprimer une i/o. L'écran 23 affiche le traitement concerné. Les fonctions de consultation du point 1.2.1. sont toujours possibles.
Pour supprimer une i/o, taper <D> et l'écran 24 vous demandera un numéro d'i/o à supprimer.
REM : il n'est pas possible de supprimer la dernière i/o d'un traitement car un traitement sans entrée et sans sortie n'a pas de sens.

1.2.3.6. L'inversion d'une i/o dans un traitement.

Inverser une i/o, au sein d'un traitement, signifie transférer du côté sortie (du côté entrée) une i/o qui était du côté entrée (du côté sortie).

La procédure initiale est similaire au point 1.2.3.5., ensuite l'écran 27 demande le numéro de l'i/o à inverser.

REM : toujours dans le but d'éviter les doubles qui n'ont aucun sens, il n'est pas possible d'inverser une i/o si la destination de son transfert (le côté entrée ou le côté sortie) comprend déjà cette i/o.

1.2.3.7. L'ajout d'un traitement dans le projet.

Sélectionner l'option 7 de l'écran 10.
L'écran 28 affiche les numéros et les noms des traitements et demande à l'utilisateur où il veut intercaler le nouveau traitement pour garder la chronologie du diagramme de flux et comment il veut appeler ce traitement (encore ici, l'utilisateur ne saura pas attribuer à ce traitement un nom qui existe déjà).
Si l'utilisateur, par exemple, donne le numéro 3 au nouveau traitement, le programme intercalera le nouveau traitement entre le deuxième et le troisième traitement.
Ensuite, une fiche de traitement vierge (écran 29) apparaît et l'utilisateur la remplira en suivant les règles énoncées au point 1.1.

1.2.3.8. La suppression d'un traitement dans le projet.

Sélectionner l'option 8 de l'écran 10.
L'écran 30 affiche les numéros et les noms des traitements. Il suffit de choisir un numéro de traitement à détruire et l'opération s'effectue.

1.2.3.9. Retour vers l'écran 6.

1.2.4. Lancement du calcul d'estimation.

Sélectionner l'option 4 de l'écran 6.
L'écran 31 affiche le nombre de points de fonction brut et demande à l'utilisateur d'attribuer une valeur de 0 à 5 aux 15 facteurs de correction. Si, en face d'un facteur, l'utilisateur tape <return>, le programme enregistre par défaut la valeur moyenne (2.5) pour ce facteur.
Une fois ces 15 facteurs pondérés, l'écran 32 affiche le nombre de points de fonction brut et net ainsi que le nombre d'inputs, outputs, master files, inquiries et d'interfaces du projet.
En pressant n'importe quelle touche, on retourne vers l'écran 6.

1.2.5. Retour au menu principal avec sauvetage du projet.

Le retour au menu principal s'effectue en sélectionnant l'option 5 de l'écran 6 mais le sauvetage ne se fait pas nécessairement automatiquement.

En effet, le programme vérifie si les i/o sont au moins utilisées une fois et si les traitements ont au moins une entrée ou une sortie. Si tout est correct, le sauvetage et le retour au menu principal s'effectuent. Si ce n'est pas le cas, l'écran 33 affiche 2 fenêtres indiquant ces anomalies et donne la possibilité à l'utilisateur de les corriger. Si l'utilisateur ne tient pas à les corriger, le sauvetage du projet et le retour au menu principal s'effectuent, sinon les fenêtres s'effacent et l'écran 6 apparaît.

2. La modification des paramètres de la méthode d'estimation .

Sélectionner l'option 2 de l'écran 1 (menu principal). L'écran 34 présente les différentes valeurs de complexité attribuées aux inputs, outputs, master files, inquiries et interfaces. Il est possible de les modifier en tapant <M>, mais attention, ce choix provoque la destruction de toutes les valeurs de complexité. Il faut donc les réintroduire toutes.

En tapant <S>, le retour au menu principal est immédiat.

3. L'historique des estimations.

Cette partie du programme n'est pas encore développée mais nous comptons la construire dans le but d'attribuer par des calculs statistiques un nombre moyen d'heures par point de fonction.

ECRAN 1

MENU PRINCIPAL

1. ESTIMATION DU COUT D'UN PROJET.
2. MODIFICATION DE PARAMETRES DANS LA METHODE D'ESTIMATION.
3. HISTORIQUE DES ESTIMATIONS.
4. ---> SORTIE DU PROGRAMME.

CHOIX : ~~

ECRAN 2.

ESTIMATION DU COUT D'UN PROJET

NOM DU PROJET : ~~

N°	NOM PROJET	N°	NOM PROJET	N°	NOM PROJET	N°	NOM PROJET
1	travellercheq						
2	nouveau						

ECRAN 3.

ESTIMATION DU COUT D'UN PROJET

NOM DU PROJET : essai — NOUVEAU PROJET

1. CREATION DE FICHES I/O.

2. ---> RETOUR AU MENU PRINCIPAL.

CHOIX :

ECRAN 4.

INTRODUCTION DES I/O DU PROJET

NOM DU PROJET : essai — PAGE 1

N°	TYPE	NOM	CPX		N°	TYPE	NOM	CPX

TYPE : | NOM DE L'ENTREE OU DE LA SORTIE : | CPX :

ECRAN 6.

ESTIMATION DU COUT D'UN PROJET

NOM DU PROJET : essai ANCIEN PROJET

1. CONSULTATION DE FICHES I/O.
2. CONSULTATION DES CARACTERISTIQUES DES I/O.
3. MODIFICATIONS DE TRAITEMENTS ET D'I/O
4. LANCEMENT DU CALCUL D'ESTIMATION.

5. ---> RETOUR AU MENU PRINCIPAL ET SAUVETAGE DU PROJET.

CHOIX :

ECRAN 6

ESTIMATION DU COUT D'UN PROJET

NOM DU PROJET : travellercheq ANCIEN PROJET

1. CONSULTATION DE FICHES I/O.
2. CONSULTATION DES CARACTERISTIQUES DES I/O.
3. MODIFICATIONS DE TRAITEMENTS ET D'I/O
4. LANCEMENT DU CALCUL D'ESTIMATION.

5. ---> RETOUR AU MENU PRINCIPAL.

CHOIX :

ECRAN 7

CONSULTATION DES FICHES I/O

NOM DU PROJET : travellercheq

N°	TRAITEMENT	N°	TRAITEMENT	N°	TRAITEMENT	N°	TRAITEMENT
1	coord1	15	comfirdeliv				
2	coord2	16	consdelstock				
3	ajout	17	adorder				
4	coord3	18	correctorder				
5	coord4	19	confirmorder				
6	coord5	20	consorder				
7	coord6	21	adsale				
8	acdemit	22	corsale				
9	ad-del-cp	23	conssale				
10	consultation	24	printsale				
11	addevies						
12	consdevies						
13	addelivery						
14	cordelivery						

CHOISISSEZ UN N° DE TRAITEMENT... ~~ <TAPER 0 POUR SORTIR>

ECRAN 8

CONSULTATION DE LA FICHE N° 8

NOM DU TRAITEMENT : acdemit

N°	TYPE	ENTREE	CPX	N°	TYPE	SORTIE	CPX
1	ed	ecr2	s	15	fd	m-emit-adr	s
2	fd	m-emit-adr	s	16	fd	d-emit-mail	s
3	fd	d-emit-mail	s				

TAPER <+> POUR PAGE SUIVANTE, <-> POUR PAGE PRECEDENTE OU <S> POUR SORTIR :

ECRAN 9

CONSULTATION DES CARACTERISTIQUES DES I/O DU PROJET

NOM DU PROJET : travellercheq | PAGE 1

N°	TYPE	NOM	CPX	N°	TYPE	NOM	CPX
1	ed	menuprincipal	s	15	?ed	ecr5	s
2	ed	ecrdevies	s	16	?d	doc2	s
3	ed	ecrdelivery	s	17	ed	ecr11	s
4	ed	ecrorder	s	18	fd	d-stock-chang	c
5	ed	ecrsale	s	19	fd	nnn	c
6	ed	ecr2	s	20	ed	ecr12	s
7	fd	m-emit-adr	s	21	d	enreg-correct	s
8	fd	d-emit-mail	s	22	ed	ecr16	s
9	fd	m-kind-tc	s	23	d	confirm	s
10	ed	ecr3	s	24	?ed	ecr10	s
11	?d	doc1	s	25	?d	ecr14	s
12	?ed	ecr7	s	26	?d	doc3	s
13	ed	ecr4	s	27	ed	ecr18	s
14	fd	rates	s	28	fd	m-nr-last-del	s

TAPER <+> POUR PAGE SUIVANTE, <-> POUR PAGE PRECEDENTE OU <S> POUR SORTIR :

ECRAN 10

ESTIMATION DU COUT D'UN PROJET

NOM DU PROJET : travellercheq — ANCIEN PROJET

1. CORRECTION DES CARACTERISTIQUES DES I/O DANS LE PROJET.
2. AJOUT D'UNE I/O DANS LE PROJET.
3. AJOUT D'UNE I/O DANS UN TRAITEMENT.
4. SUPPRESSION D'UNE I/O DANS LE PROJET.
5. SUPPRESSION D'UNE I/O DANS UN TRAITEMENT.
6. INVERSION D'UNE I/O DANS UN TRAITEMENT.
7. AJOUT D'UN TRAITEMENT.
8. SUPPRESSION D'UN TRAITEMENT.
9. ---> SORTIE.

CHOIX :

ECRAN 11

CORRECTION DES CARACTERISTIQUES DES I/O DU PROJET

NOM DU PROJET : travellercheq — PAGE 1

N°	TYPE	NOM	CPX	N°	TYPE	NOM	CPX
1	ed	menuprincipal	s	15	?ed	ecr5	s
2	ed	ecrdevies	s	16	?d	doc2	s
3	ed	ecrdelivery	s	17	ed	ecr11	s
4	ed	ecrorder	s	18	fd	d-stock-chang	c
5	ed	ecrsale	s	19	fd	nnn	c
6	ed	ecr2	s	20	ed	ecr12	s
7	fd	m-emit-adr	s	21	d	enreg-correct	s
8	fd	d-emit-mail	s	22	ed	ecr16	s
9	fd	m-kind-tc	s	23	d	confirm	s
10	ed	ecr3	s	24	?ed	ecr10	s
11	?d	doc1	s	25	?d	ecr14	s
12	?ed	ecr7	s	26	?d	doc3	s
13	ed	ecr4	s	27	ed	ecr18	s
14	fd	rates	s	28	fd	m-nr-last-del	s

<+> = PAGE SUIVANTE, <-> = PAGE PRECEDENTE, <C> = CORRIGER, <S> = SORTIR :

ECRAN 12

CORRECTION DES CARACTERISTIQUES DES I/O DU PROJET

NOM DU PROJET : travellercheq — PAGE 1

N°	TYPE	NOM	CPX	N°	TYPE	NOM	CPX
1	ed	menuprincipal	s	15	?ed	ecr5	s
2	ed	ecrdevies	s	16	?d	doc2	s
3	ed	ecrdelivery	s	17	ed	ecr11	s
4	ed	ecrorder	s	18	fd	d-stock-chang	c
5	ed	ecrsale	s	19	fd	nnn	c
6	ed	ecr2	s	20	ed	ecr12	s
7	fd	m-emit-adr	s	21	d	enreg-correct	s
8	fd	d-emit-mail	s	22	ed	ecr16	s
9	fd	m-kind-tc	s	23	d	confirm	s
10	ed	ecr3	s	24	?ed	ecr10	s
11	?d	doc1	s	25	?d	ecr14	s
12	?ed	ecr7	s	26	?d	doc3	s
13	ed	ecr4	s	27	ed	ecr18	s
14	fd	rates	s	28	fd	m-nr-last-del	s

Quel est le n° de l'entrée ou de la sortie à modifier ?

ECRAN 13

CORRECTION DES CARACTERISTIQUES DES I/O DU PROJET

NOM DU PROJET : travellercheq — PAGE 1

N°	TYPE	NOM	CPX	N°	TYPE	NOM	CPX
1	ed	menuprincipal	s	15	?ed	ecr5	s
2	ed	ecrdevies	s	16	?d	doc2	s
3	ed	ecrdelivery	s	17	ed	ecr11	s
4	ed	ecrorder	s	18	fd	d-stock-chang	c
5	ed	ecrsale	s	19	fd	nnn	c
6	ed	ecr2	s	20	ed	ecr12	s
7	fd	m-emit-adr	s	21	d	enreg-correct	s
8	fd	d-emit-mail	s	22	ed	ecr16	s
9	fd	m-kind-tc	s	23	d	confirm	s
10	ed	ecr3	s	24	?ed	ecr10	s
11	?d	doc1	s	25	?d	ecr14	s
12	?ed	ecr7	s	26	?d	doc3	s
13	ed	ecr4	s	27	ed	ecr18	s
14	fd	rates	s	28	fd	m-nr-last-del	s

TYPE : | NOM DE L'ENTREE OU DE LA SORTIE : | CPX :

ECRAN 14

AJOUT D' I/O DANS LE PROJET

NOM DU PROJET : travellercheq — PAGE 1

N°	TYPE	NOM	CPX	N°	TYPE	NOM	CPX
1	ed	menuprincipal	s	15	?ed	ecr5	s
2	ed	ecrdevies	s	16	?d	doc2	s
3	ed	ecrdelivery	s	17	ed	ecr11	s
4	ed	ecrorder	s	18	fd	d-stock-chang	c
5	ed	ecrsale	s	19	fd	nnn	c
6	ed	ecr2	s	20	ed	ecr12	s
7	fd	m-emit-adr	s	21	d	enreg-correct	s
8	fd	d-emit-mail	s	22	ed	ecr16	s
9	fd	m-kind-tc	s	23	d	confirm	s
10	ed	ecr3	s	24	?ed	ecr10	s
11	?d	doc1	s	25	?d	ecr14	s
12	?ed	ecr7	s	26	?d	doc3	s
13	ed	ecr4	s	27	ed	ecr18	s
14	fd	rates	s	28	fd	m-nr-last-del	s

<+> = PAGE SUIVANTE, <-> = PAGE PRECEDENTE, <A> = AJOUT, <S> = SORTIR :

ECRAN 15

AJOUT D' I/O DANS LE PROJET

NOM DU PROJET : travellercheq — PAGE 2

N°	TYPE	NOM	CPX	N°	TYPE	NOM	CPX
29	fd	d-stock-agenc	c	43	d	enreg-sale	s
30	fd	rebu	s	44	ed	ecr26	s
31	d	enreg-order	s	45	d	enreg-cor-sal	s
32	ed	ecr19	s	46	?ed	ecr27	s
33	d	enreg-cor-ord	s	47	?d	doc5	s
34	ed	ecr20	s	48	?ed	ecr28	s
35	fd	a-nr-delivery	s	49	ed	ecr29	s
36	?ed	ecr21	s	50	fc	printbatch	m
37	?d	doc4	s	51	?ed	ecr6	s
38	?ed	ecr22	s	52	?ed	ecr	s
39	ed	ecr25	s	53	d	rap	c
40	fd	a-nr-agency	s				
41	fd	d-sold-tc	c				
42	fd	d-amount-deli	m				

TYPE : | NOM DE L'ENTREE OU DE LA SORTIE : | CPX :

ECRAN 16.

AJOUT D'I/O DANS UN TRAITEMENT

NOM DU PROJET : travellercheq

N°	TRAITEMENT	N°	TRAITEMENT	N°	TRAITEMENT	N°	TRAITEMENT
1	coord1	15	comfirdeliv				
2	coord2	16	consdelstock				
3	ajout	17	adorder				
4	coord3	18	correctorder				
5	coord4	19	confirmorder				
6	coord5	20	consorder				
7	coord6	21	adsale				
8	acdemit	22	corsale				
9	ad-del-cp	23	conssale				
10	consultation	24	printsale				
11	addevies						
12	consdevies						
13	addelivery						
14	cordelivery						

CHOISISSEZ UN N° DE TRAITEMENT... <TAPER 0 POUR SORTIR>

ECRAN 17

AJOUT D'UNE I/O DANS LA FICHE N° 24

NOM DU TRAITEMENT : printsale

N°	TYPE	ENTREE	CPX		N°	TYPE	SORTIE	CPX
1	ed	ecr29	s		15	fc	printbatch	m
2	fd	d-sold-tc	c		16	ed	ecrsale	s
3	fd	d-amount-deli	m					

<+> = PAGE SUIVANTE, <-> = PAGE PRECEDENTE, <A> = AJOUT, <S> = SORTIR :

ECRAN 18

AJOUT D'UNE I/O DANS LA FICHE N° 24

NOM DU TRAITEMENT : printsale

N°	TYPE	ENTREE	CPX
1	ed	ecr29	s
2	fd	d-sold-tc	c
3	fd	d-amount-deli	m

N°	TYPE	SORTIE	CPX
15	fc	printbatch	m
16	ed	ecrsale	s

NOM D'UNE ENTREE OU D'UNE SORTIE : ~

TAPER <E> OU <S> :

ECRAN 19

LISTE DES I/O

6. ecr2
7. m-emit-adr
8. d-emit-mail
9. m-kind-tc
10. ecr3
11. doc1
12. ecr7
13. ecr4
14. rates
15. ecr5

D'UNE I/O DANS LA FICHE N° 24

rintsale

N			CPX
			s
			c
		eli	m

N°	TYPE	SORTIE	CPX
15	fc	printbatch	m
16	ed	ecrsale	s

NOM D'UNE ENTREE OU D'UNE SORTIE : ?

TAPER <E> OU <S> :

ECRAN 20

SUPPRESSION D' I/O DANS LE PROJET

NOM DU PROJET : travellercheq — PAGE 1

N°	TYPE	NOM	CPX	N°	TYPE	NOM	CPX
1	ed	menuprincipal	s	15	?ed	ecr5	s
2	ed	ecrdevies	s	16	?d	doc2	s
3	ed	ecrdelivery	s	17	ed	ecr11	s
4	ed	ecrorder	s	18	fd	d-stock-chang	c
5	ed	ecrsale	s	19	fd	nnn	c
6	ed	ecr2	s	20	ed	ecr12	s
7	fd	m-emit-adr	s	21	d	enreg-correct	s
8	fd	d-emit-mail	s	22	ed	ecr16	s
9	fd	m-kind-tc	s	23	d	confirm	s
10	ed	ecr3	s	24	?ed	ecr10	s
11	?d	doc1	s	25	?d	ecr14	s
12	?ed	ecr7	s	26	?d	doc3	s
13	ed	ecr4	s	27	ed	ecr18	s
14	fd	rates	s	28	fd	m-nr-last-del	s

<+> = PAGE SUIVANTE, <-> = PAGE PRECEDENTE, <D> = DETRUIRE, <S> = SORTIR :

ECRAN 21

SUPPRESSION D' I/O DANS LE PROJET

NOM DU PROJET : travellercheq — PAGE 1

N°	TYPE	NOM	CPX	N°	TYPE	NOM	CPX
1	ed	menuprincipal	s	15	?ed	ecr5	s
2	ed	ecrdevies	s	16	?d	doc2	s
3	ed	ecrdelivery	s	17	ed	ecr11	s
4	ed	ecrorder	s	18	fd	d-stock-chang	c
5	ed	ecrsale	s	19	fd	nnn	c
6	ed	ecr2	s	20	ed	ecr12	s
7	fd	m-emit-adr	s	21	d	enreg-correct	s
8	fd	d-emit-mail	s	22	ed	ecr16	s
9	fd	m-kind-tc	s	23	d	confirm	s
10	ed	ecr3	s	24	?ed	ecr10	s
11	?d	doc1	s	25	?d	ecr14	s
12	?ed	ecr7	s	26	?d	doc3	s
13	ed	ecr4	s	27	ed	ecr18	s
14	fd	rates	s	28	fd	m-nr-last-del	s

Quel est le n° de l'entrée ou de la sortie à détruire ?

ECRAN 22

SUPPRESSION D'I/O D'UN TRAITEMENT

NOM DU PROJET : travellercheq

N°	TRAITEMENT	N°	TRAITEMENT	N°	TRAITEMENT	N°	TRAITEMENT
1	coord1	15	comfirdeliv				
2	coord2	16	consdelstock				
3	ajout	17	adorder				
4	coord3	18	correctorder				
5	coord4	19	confirmorder				
6	coord5	20	consorder				
7	coord6	21	adsale				
8	acdemit	22	corsale				
9	ad-del-cp	23	conssale				
10	consultation	24	printsale				
11	addevies						
12	consdevies						
13	addelivery						
14	cordelivery						

CHOISISSEZ UN N° DE TRAITEMENT... <TAPER 0 POUR SORTIR>

ECRAN 23

SUPPRESSION D'UNE I/O DANS LA FICHE N° 22

NOM DU TRAITEMENT : corsale

N°	TYPE	ENTREE	CPX		N°	TYPE	SORTIE	CPX
1	ed	ecr26	s		15	fd	d-stock-agenc	c
2	fd	a-nr-agency	s		16	fd	d-sold-tc	c
3	fd	d-stock-agenc	c		17	fd	d-amount-deli	m
4	fd	d-sold-tc	c		18	d	enreg-cor-sal	s
5	fd	m-nr-last-del	s		19	ed	ecrsale	s
6	fd	d-amount-deli	m					
7	fd	rebu	s					
8	fd	rates	s					

<+> = PAGE SUIVANTE, <-> = PAGE PRECEDENTE, <D> = DETRUIRE, <S> = SORTIR :

ECRAN 24

SUPPRESSION D'UNE I/O DANS LA FICHE N° 22

NOM DU TRAITEMENT : corsale

N°	TYPE	ENTREE	CPX
1	ed	ecr26	s
2	fd	a-nr-agency	s
3	fd	d-stock-agenc	c
4	fd	d-sold-tc	c
5	fd	m-nr-last-del	s
6	fd	d-amount-deli	m
7	fd	rebu	s
8	fd	rates	s

N°	TYPE	SORTIE	CPX
15	fd	d-stock-agenc	c
16	fd	d-sold-tc	c
17	fd	d-amount-deli	m
18	d	enreg-cor-sal	s
19	ed	ecrsale	s

Quel est le numéro de l'I/O à supprimer ?

ECRAN 25

INVERSION D'I/O DANS UN TRAITEMENT

NOM DU PROJET : travellercheq

N°	TRAITEMENT	N°	TRAITEMENT	N°	TRAITEMENT	N°	TRAITEMENT
1	coord1	15	comfirdeliv				
2	coord2	16	consdelstock				
3	ajout	17	adorder				
4	coord3	18	correctorder				
5	coord4	19	confirmorder				
6	coord5	20	consorder				
7	coord6	21	adsale				
8	acdemit	22	corsale				
9	ad-del-cp	23	conssale				
10	consultation	24	printsale				
11	addevies						
12	consdevies						
13	addelivery						
14	cordelivery						

CHOISISSEZ UN N° DE TRAITEMENT... <TAPER O POUR SORTIR>

ECRAN 26.

INVERSION D'UNE I/O DANS LA FICHE N° 22

NOM DU TRAITEMENT : corsale

N°	TYPE	ENTREE	CPX	N°	TYPE	SORTIE	CPX
1	ed	ecr26	s	15	fd	d-stock-agenc	c
2	fd	a-nr-agency	s	16	fd	d-sold-tc	c
3	fd	d-stock-agenc	c	17	d	enreg-cor-sal	s
4	fd	d-sold-tc	c	18	ed	ecrsale	s
5	fd	m-nr-last-del	s				
6	fd	d-amount-deli	m				
7	fd	rebu	s				
8	fd	rates	s				

<+> = PAGE SUIVANTE, <-> = PAGE PRECEDENTE, <I> = INVERSER, <S> = SORTIR :

ECRAN 27

INVERSION D'UNE I/O DANS LA FICHE N° 22

NOM DU TRAITEMENT : corsale

N°	TYPE	ENTREE	CPX	N°	TYPE	SORTIE	CPX
1	ed	ecr26	s	15	fd	d-stock-agenc	c
2	fd	a-nr-agency	s	16	fd	d-sold-tc	c
3	fd	d-stock-agenc	c	17	d	enreg-cor-sal	s
4	fd	d-sold-tc	c	18	ed	ecrsale	s
5	fd	m-nr-last-del	s				
6	fd	d-amount-deli	m				
7	fd	rebu	s				
8	fd	rates	s				

QUEL EST LE NUMERO DE L'I/O A INVERSER ?

ECRAN 28

AJOUT D'UN TRAITEMENT

NOM DU PROJET : travellercheq

N°	TRAITEMENT	N°	TRAITEMENT	N°	TRAITEMENT	N°	TRAITEMENT
1	coord1	15	comfirdeliv				
2	coord2	16	consdelstock				
3	ajout	17	adorder				
4	coord3	18	correctorder				
5	coord4	19	confirmorder				
6	coord5	20	consorder				
7	coord6	21	adsale				
8	acdemit	22	corsale				
9	ad-del-cp	23	conssale				
10	consultation	24	printsale				
11	addevies						
12	consdevies						
13	addelivery						
14	cordelivery						

SON NUMERO (OU 0 POUR SORTIR): 25 NOM DU NOUVEAU TRAITEMENT :

AJOUT D'UN TRAITEMENT

NOM DU PROJET : travellercheq

N°	TRAITEMENT	N°	TRAITEMENT	N°	TRAITEMENT	N°	TRAITEMENT
1	coord1	15	comfirdeliv				
2	coord2	16	consdelstock				
3	ajout	17	adorder				
4	coord3	18	correctorder				
5	coord4	19	confirmorder				
6	coord5	20	consorder				
7	coord6	21	adsale				
8	acdemit	22	corsale				
9	ad-del-cp	23	conssale				
10	consultation	24	printsale				
11	addevies						
12	consdevies						
13	addelivery						
14	cordelivery						

SON NUMERO (OU 0 POUR SORTIR): NOM DU NOUVEAU TRAITEMENT :

ECRAN 29

AJOUT D'UN TRAITEMENT

NOM DU TRAITEMENT : tri

N°	TYPE	ENTREE	CPX		N°	TYPE	SORTIE	CPX

NOM D'UNE ENTREE OU D'UNE SORTIE : TAPER <E> OU <S> :

ECRAN 30.

SUPPRESSION D'UN TRAITEMENT

NOM DU PROJET : travellercheq

N°	TRAITEMENT	N°	TRAITEMENT	N°	TRAITEMENT	N°	TRAITEMENT
1	tri	15	cordelivery				
2	coord1	16	comfirdeliv				
3	coord2	17	consdelstock				
4	ajout	18	adorder				
5	coord3	19	correctorder				
6	coord4	20	confirmorder				
7	coord5	21	consorder				
8	coord6	22	adsale				
9	acdemit	23	corsale				
10	ad-del-cp	24	conssale				
11	consultation	25	printsale				
12	addevies						
13	consdevies						
14	addelivery						

CHOISISSEZ UN N° DE TRAITEMENT... <TAPER 0 POUR SORTIR>

LES POINTS DE FONCTION

NOM DU PROJET : travellercheq

---> LE NOMBRE DE POINTS DE FONCTION BRUT = 671 <---

Attribuez une valeur de 0 à 5 à chacun de ces 15 facteurs :

1. l'utilisation d'un logiciel DB/DC.....
2. l'application est développée pour plusieurs utilisateurs.....
3. l'aspect performance est important.....
4. l'ordinateur est fort sollicité.....
5. l'application est fortement transactionnelle.....
6. l'application est du genre "on line data entry".....
7. l'application réalise la plupart des fonctions en direct.....
8. la mise à jour des données est faite en direct.....
9. les processus sont particulièrement complexes.....
10. l'application comprend des parties utilisées par d'autres projets...
11. la phase de travaux de conversion et/ou de démarrage est importante.
12. le degré d'automaticité du système est fort important.....
13. l'application sera utilisée en des sites différents.....
14. l'application permet une évolution importante et des changements....
15. la disponibilité du matériel.....

ECRAN 31

LES POINTS DE FONCTION

NOM DU PROJET : travellercheq

---> LE NOMBRE DE POINTS DE FONCTION BRUT = 671 <---

Attribuez une valeur de 0 à 5 à chacun de ces 15 facteurs :

1. l'utilisation d'un logiciel DB/DC..... 2.5
2. l'application est développée pour plusieurs utilisateurs..... 2.5
3. l'aspect performance est important..... 2
4. l'ordinateur est fort sollicité..... 5
5. l'application est fortement transactionnelle..... 0
6. l'application est du genre "on line data entry"..... 3
7. l'application réalise la plupart des fonctions en direct..... 4
8. la mise à jour des données est faite en direct..... 2.5
9. les processus sont particulièrement complexes..... 2.5
10. l'application comprend des parties utilisées par d'autres projets...
11. la phase de travaux de conversion et/ou de démarrage est importante.
12. le degré d'automaticité du système est fort important.....
13. l'application sera utilisée en des sites différents.....
14. l'application permet une évolution importante et des changements....
15. la disponibilité du matériel.....

ECRAN 32

RESULTATS DE LA METHODE DES POINTS DE FONCTION

NOM DU PROJET : travellercheq

LE NOMBRE DE POINTS DE FONCTION BRUT = 671

LE NOMBRE DE POINTS DE FONCTION NET = 681

POUR : 82 INPUTS
36 OUTPUTS
0 MASTER FILES
10 INQUIRIES
21 INTERFACES

ET AVEC UNE CORRECTION DE : ± 1 %

TAPEZ UNE TOUCHE

ECRAN 33

ATTENTION

Ces traitements n'ont pas d'i/o

1. trt2

ATTENTION

Ces i/o ne sont pas utilisées

1. rap
2. stat
3. fichcli
4. doc
5. ecr2

1. CONSULTATION DE FICHES I/O.
2. CONSULTATION DES CARACTERIST
3. MODIFICATIONS DE TRAITEMENTS ET D'I/O
4. LANCEMENT DU CALCUL D'ESTIMATION.
5.

Voulez-vous corriger ces anomalies, tapez <O> ou <N> ?

: 5

ÉCRAN 34.

MAINTENANCE DES PARAMETRES DE LA METHODE

TAPER <M> MODIFIER LES PARAMETRES OU <S> POUR SORTIR :

AUX INPUTS		AUX OUTPUTS		AUX INTERFACES	
- simples	: 3	- simples	: 4	- simples	: 5
- moyennes	: 4	- moyennes	: 5	- moyennes	: 7
- complexes	: 6	- complexes	: 7	- complexes	: 10

AUX INQUIRIES		AUX MASTERFILES	
- simples	: 3	- simples	: 7
- moyennes	: 4	- moyennes	: 10
- complexes	: 6	- complexes	: 15